PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :
G06F 9/46

A1

(11) International Publication Number: WO 98/02810

(43) International Publication Date: 22 January 1998 (22.01.98)

(21) International Application Number: PCT/US97/11883

(22) International Filing Date: 10 July 1997 (10.07.97)

(30) Priority Data:
08/680,203 11 July 1996 (11.07.96) US

(71) Applicant: TANDEM COMPUTERS INCORPORATED [US/US]; 10435 N. Tantau Avenue, LOC. 200-16, Cupertino, CA 95014 (US).

(72) Inventor: SCHOFIELD, Andrew; Lindenbuehl 27, CH-6330 Cham (US).

(74) Agents: GRANATELLI, Lawrence, W. et al.; 600 Hansen Way, Palo Alto, CA 94304 (US).

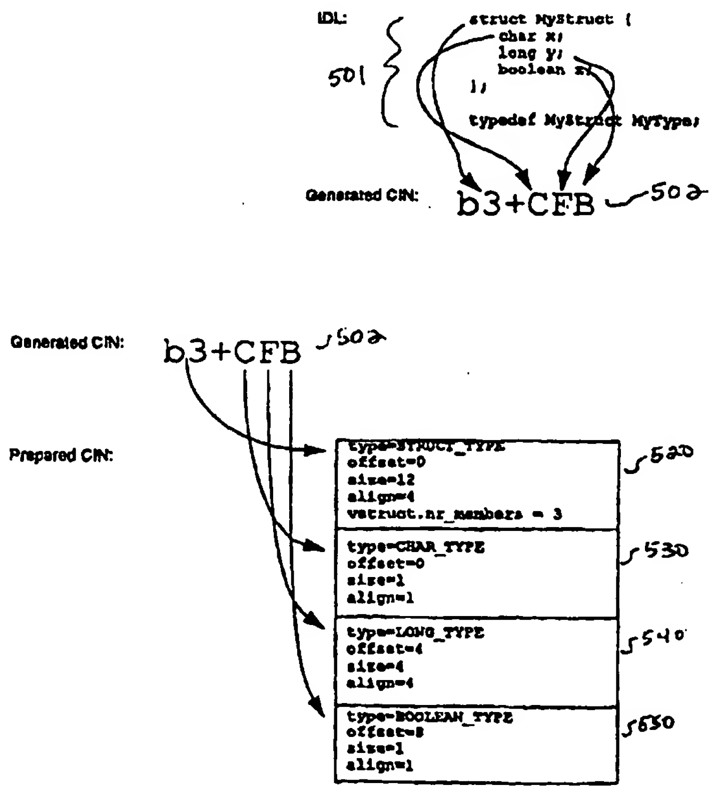
(81) Designated States: JP, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: METHOD AND APPARATUS FOR TRANSPORTING INTERFACE DEFINITION LANGUAGE-DEFINED DATA STRUCTURES BETWEEN HETEROGENEOUS SYSTEMS

(57) Abstract

A method and apparatus for transporting IDL-defined data structures to and from a format convenient for transport between two computers are disclosed. The data structures are originally described in a string. The string description is converted to a different format containing additional information about the alignment and size of the data structure. An application in the sending computer removes the alignment form the data structure and stores the data structure in a buffer. The data structure is stored in the output buffer in a predetermined format that is based upon the type of the data structure. The buffer is then transmitted to a data file or to the memory of the receiving computer. The receiving computer extracts the data structure from the buffer based upon the predetermined format. The data structure is realigned and stored in the memory of the second computer.



```
struct MyStruct {
    char x;
    long y;
    boolean z;
};

typedef MyStruct MyType;
```

```
type=STRUCT_TYPE
offset=0
size=12
align=4
vstruct.nr_members = 3
```

```
type=CHAR_TYPE
offset=0
size=1
align=1
```

```
type=LONG_TYPE
offset=4
size=4
align=4
```

```
type=BOOLEAN_TYPE
offset=8
size=1
align=1
```

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece	ML	Mali	TR	Turkey
BG	Bulgaria	HU	Hungary	MN	Mongolia	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MR	Mauritania	UA	Ukraine
BR	Brazil	IL	Israel	MW	Malawi	UG	Uganda
BY	Belarus	IS	Iceland	MX	Mexico	US	United States of America
CA	Canada	IT	Italy	NE	Niger	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NL	Netherlands	VN	Viet Nam
CG	Congo	KE	Kenya	NO	Norway	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NZ	New Zealand	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	PL	Poland		
CM	Cameroon	KR	Republic of Korea	PT	Portugal		
CN	China	KZ	Kazakstan	RO	Romania		
CU	Cuba	LC	Saint Lucia	RU	Russian Federation		
CZ	Czech Republic	LI	Liechtenstein	SD	Sudan		
DE	Germany	LK	Sri Lanka	SE	Sweden		
DK	Denmark	LR	Liberia	SG	Singapore		
EE	Estonia						

METHOD AND APPARATUS FOR TRANSPORTING INTERFACE DEFINITION LANGUAGE-DEFINED DATA STRUCTURES BETWEEN HETEROGENEOUS SYSTEMS

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a method and apparatus for transporting data structures described in the Object Management Group's Interface Definition Language between heterogeneous platforms. More particularly, the present invention utilizes functions for removing the alignment from data structures and storing the data structures in a predetermined format for transport to a file or across heterogeneous platforms.

2. Background

Distributed object computing combines the concepts of distributed computing and object-oriented computing. Distributed computing consists of two or more pieces of software sharing information with each other. These two pieces of software could be running on the same computer or on different computers connected to a common network. Most distributed computing is based on a client/server mode. With the client/server model, two major types of software are utilized: client software, which requests the information or service, and server software, which provides the information or service.

Object-oriented computing is based upon the object model where pieces of code called "objects"--often abstracted from real objects in the real world--own data (called "attributes" in object-oriented programming parlance) and provide services through methods (also known as "operations" or "member functions"). The data and methods contained in an object may be "public" or "private." Public data may be altered by any other object. Most data, however, is private and accessible only to methods owned by the object. Typically, the methods operate on the private data contained in the object.

A collection of similar objects make up an interface (or "class" in C++ parlance). An interface specifies the methods and types of data contained in all objects of the interface. Objects are then created ("instantiated") based upon that interface. Each object contains data specific to that object. Each specific object is identified within a distributed object system by a unique identifier called an object reference.

In a distributed object system, a client sends a request (or "object call") containing an indication of the operation for the server to perform, the object reference, and a mechanism to return "exception information" (unexpected occurrences) about the success or failure of a request. The server receives the request and, if possible, carries out the request and returns the appropriate exception information. An object request broker ("ORB") provides a communication hub for all objects in the system passing the request to the server and returning the reply to the client.

On the client side, the ORB handles requests for the invocation of a method and the related selection of servers and methods. When an application sends a request to the ORB for a method to be performed on an object, the ORB validates the arguments contained in the request against the interface and dispatches the request to the server, starting it if necessary. On the server side, the ORB receives such requests, unmarshals the arguments, sets up the context state as needed, invokes the method dispatcher, marshals the output arguments, and returns the results to the client, thereby completing the object invocation.

Both client and server must have information about the available objects and methods that can be performed. Through the hiding of private data ("encapsulation" in object-oriented parlance), the client does not need to know how the request will be carried out by the server. Nevertheless, both client and server must have access to common interface definitions to enable communication therebetween. Currently, the standard language for distributed object computing is the Object Management Group's ("OMG") Interface Definition Language ("IDL").

A distributed object system developer defines the system's available interfaces in IDL. An interface includes one or more operations that can be performed on objects of that interface. Each operation may receive one or more parameters. Each parameter is of a particular IDL data type.

IDL includes several data types. Integers are represented through long and short, signed and unsigned integer data types. OMG IDL floating point types are float and double. The float type represents IEEE single-precision floating point numbers while the double type represents the IEEE double-precision floating point numbers. OMG IDL defines a char data type consisting of 8-bit quantities. The boolean data type is used to represent true/false values. The "any" type permits the specification of values that can express any OMG IDL type. Complex types, such as structures, unions, and templates are also represented.

IDL is designed to be used in distributed object systems implementing OMG's Common Object Request Broker Architecture ("CORBA"). In a typical CORBA system, interface definitions are written in an IDL-defined source file (also known as a "translation unit"). The source file is compiled by an IDL compiler that maps the source file to a specific programming language. The IDL compiler generates programming-language-specific files, including client stub files, header files, and server skeleton files. Client stub files are then compiled and linked into client applications and are used to make requests. Header files are linked into client and server applications and are used to define data types. Server skeleton files are linked into server applications and are used to map client operations on objects (requests) to methods in a server implementation.

When object calls are made, data structures are transported from one computer system to another (client-to-server and server-to-client). Such object calls may occur between identical systems, but are likely to be made across heterogeneous platforms using different operating systems, programming languages, and compilers. Once IDL source files have been compiled and mapped to a particular programming language, each independent compiler vendor will align data structures on the stack in a particular manner. Accordingly, both the client and the server systems may align data structures differently. Moreover, both the client systems may align parameters within a structure differently. If the client and server application do not understand each other's method of alignment, the transported data structure will become garbled and an error will occur.

In addition, the hardware utilized by the client and server may be different. For example, one computer may include a CPU that requires so-called "BIGendian" integer representation where the most significant byte is listed first. The other computer may include a CPU that requires "LITTLEendian" representation where the least significant byte is listed first. Data structures cannot be passed effectively between these two machines without reciprocal knowledge of each system.

Moreover, if clients and servers are required to have detailed knowledge of each other, a primary goal of object-oriented computing--encapsulation--is lost. Both the client and the server applications must write detailed alignment functions to ensure compatibility during object calls. This extra coding work makes distributed object computing inefficient.

Accordingly, there is a need for a method for converting IDL-defined data structures into a platform-independent format, such that converted data types can be transported effectively across heterogeneous systems.

In addition, there is a need for a method that reduces the amount of code and execution time required by client and server applications to implement object calls.

SUMMARY OF THE INVENTION

The present invention is directed to a method that satisfies the need to convert IDL-defined data structures into a platform-independent format, such that converted data structures can be transported across a network. The need for reducing the amount of code and execution time required by client and server applications is also satisfied. Specifically, both systems have a description of a data structure defined in ASCII string format called the Compact IDL Notation ("CIN"). The CIN is converted to a "prepared CIN" format containing additional information about the offset and size of the data structure. The prepared CIN is used to extract data structure and store the data into an output buffer based upon the size and offset of the data structure. The data is densely packed into the output buffer without any alignment padding fields. The output buffer is transferred to an input buffer of the second system. The second computer extracts the data from its input buffer. The data is then converted to the format of the second system and realigned in a data structure based upon the prepared CIN.

The method is performed through the use of calls to implementation libraries made by the client and server systems. The Implementation Libraries include generic functions located in a run-time library. By calling generic functions, the client and server applications can easily make object calls without specific knowledge of each other and without requiring additional code.

A more complete understanding of the conversion method will be afforded to those skilled in the art, as well as a realization of additional advantages and objects thereof, by a consideration of the following detailed description of the preferred embodiment. Reference will be made to the appended sheets of drawings which will first be described briefly.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a diagram of a distributed computing environment using the method or apparatus of the present invention.

Fig. 2 is a diagram of the Common Execution Environment infrastructure.

Fig. 3 is a diagram of conventional IDL source file compilation and linking.

Fig. 4 is a diagram of IDL source file compilation and linking utilizing the method of the present invention.

Figs. 5(a) and 5(b) show an IDL-defined data structure, generated CIN description, and generated array of op_tag structures.

Fig. 6 is a flow chart describing a first preferred embodiment of the client side of the method of the present invention.

Fig. 7 is a flow chart describing a first preferred embodiment of the server side of the method of the present invention.

Fig. 8 is a flow chart depicting the generation of a CIN descriptor for base and compound data types.

Fig. 9 is a flow chart depicting the generation of a CIN descriptor for an operation.

Fig. 10 is a flow chart depicting the generation of a CIN descriptor for an interface.

Fig. 11 is a diagram showing a PIF data structure.

Fig. 12 is a diagram showing an entry data structure.

Fig. 13 is a diagram showing an operation data structure.

Fig. 14 is a diagram showing a union data structure.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

I. Hardware Overview

Reference will now be made in detail to the preferred embodiments of the invention, examples of which are illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts.

As illustrated in Figure 1, the method of the present invention is designed for use in a distributed (client/server) computing environment 10. The client and server systems are connected by network connections 12, such as internet connections or the connections of a local area network. The server computer 11 communicates over a bus of I/O channel 20 with an associated storage subsystem 13. The server system 11 includes a CPU 15 and a memory 17 for storing current state information about program execution. A portion of the memory 17 is dedicated to storing the states and variables associated with each function of the program which is currently executing on the client computer. The client computer 21 similarly includes a CPU 27 and associated memory 23, and an input device 29, such as a keyboard or a mouse and a display device 33, such as a video display terminal ("VDT"). The client CPU communicates over a bus or I/O channel 40 with a disk storage subsystem 33 and via I/O channel 41 with the keyboard 29, VDT 33 and mouse 31. Both computers are capable of reading various types of media, including floppy disks and CD-ROMs.

The client/server model as shown in Figure 1 is merely demonstrative of a typical client/server system. Within the context of the present invention, the "client" is an application that requests services while the "server" is an application that implements the requested service. Indeed, both the client and server application may reside on the same computer and within a common capsule, as discussed below. The client and server application may also reside on separate computers using different operating systems.

II. Distributed Computing Environment

The method and apparatus of the present invention may be utilized within any distributed computing environment. In a preferred embodiment, the Common Execution Environment ("CEE"), which is a component of the Tandem Message Switching Facility ("MSF") Architecture is utilized. The CEE activates and deactivates objects and is used to pass messages between client and server applications loaded in CEE capsules. The CEE may be stored in the memory of a single machine. The CEE and client and server applications may, however, be loaded on multiple machines across a network as shown in Figure 1. The client-side CEE 75 is stored in the client memory 27. The server-side CEE 80 is stored in server memory 17.

The CEE uses a "capsule" infrastructure. A capsule encapsulates memory space and execution stream. A capsule may be implemented differently on different systems depending upon the operating system. For instance, on certain systems, a capsule may be implemented as a process. On other systems, the capsule may be implemented as a thread. Moreover, client and server applications may be configured within different capsules contained on different machines as shown in Figure 1. Alternatively, the different capsules may be configured as shown in Figure 2. Figure 2a shows a client application 77 loaded in a single capsule 81 and a server application 87 may be loaded in a separate capsule 85. Both capsules, however, are stored on the same machine 21. Both the client and server applications may also be loaded within a single capsule 81 on the same machine 21 as shown in Figure 2b. As stated above, the method of the present invention will be described with reference to the multiple capsule, multiple machine case. Accordingly, the client 12 and server machine 11 include a client-side CEE 75 and a server-side CEE 85 loaded in their respective memories.

Figure 3 shows a CEE capsule 70 contained, for example, in a client computer memory 27 (not shown) that includes the CEE 75 and certain of the core CEE components and implementations of objects contained within Implementation Libraries 71. The Implementation Libraries 71 include the client application 79 (or the server application in the case of the server capsule) and client stubs 77 (or server stubs) generated from the IDL specification of the object's interface, as described below. The Implementation Libraries 71 and the CEE 75 interact through the down-calling of dynamically-accessible routines supplied by the CEE and the up-calling of routines contained in the Implementation Library. The CEE 75 can also receive object calls 82 from other capsules within the same machine and requests 84 from other CEE's. The client-side CEE 75 and the server-side CEE 85 may communicate using any known networking protocol. The client and server CEE's includes numerous libraries of routines that can be down-called from client and server applications. The Presentation Conversion Utilities ("PCU") 89 is a library of routines utilized in the method of the present invention.

Objects implemented in a CEE capsule may be configured or dynamic. Configured objects have their implementation details stored in a repository (such as the MSF Warehouse 85) or in initialization scripts. Given a request for a specific object reference, the CEE 75 starts the appropriate capsule based on this configuration data. The capsule uses the configuration data to determine which implementation library to load and which object initialization routine to call. The object initialization routine then creates the object. Dynamic objects are created and destroyed dynamically within the same capsule. Dynamic objects lack repository-stored or scripted configuration information.

The following paragraphs describe a system-level view of how the Implementation Libraries interact with the CEE 75. The CEE 75 implements requests to activate and deactivate objects within a capsule. In addition, the CEE facilitates inter-capsule object calls 82 as well as requests from other CEE's 84, as discussed above. Object activation requests arise when an object call from a client or server application must be satisfied. To activate an object, the CEE 75 loads the appropriate Implementation Library (if not already loaded) containing the object's methods and then calls a configured object initialization routine. The initialization routine specifies which interface the Implementation Libraries support and registers the entry points of the object's methods to be called by the CEE at a later time.

When the client and server systems start, both the client-side and server-side CEE's run their own initialization. This initialization tells client and server CEE's where to locate the various Implementation Libraries. Once located by the CEE, the initialization routines in the client and server applications are called. The initialization routines contained in the client and server applications must first carry out any required application-specific initialization. Next, both the client and server initialization routines call a stub function which, in turn, down-calls a CEE function (contained in a dynamic library as stated above) called CEE_INTERFACE_CREATE to specify the object's interface. An interface may be specified for each object. The interface description is normally generated from an IDL description of the interface, as discussed below. CEE_INTERFACE_CREATE creates an interface and returns an "interface handle" to the newly created interface. The handle is a unique identifier that specifies the interface. The server application initialization routine then uses the interface handle to down-call CEE_IMPLEMENTATION_CREATE. CEE_IMPLEMENTATION_CREATE creates an implementation description that can be used by one or more objects. CEE_IMPLEMENTATION_CREATE returns an "implementation handle" that is a unique identifier specifying the implementation for each operation in the interface. Finally, the server application initialization routine uses the implementation handle to call a stub function which down-calls CEE_SET_METHOD. CEE_SET_METHOD specifies the actual addresses of specific method routines of the implementation as contained in the server application. The CEE then has sufficient information to connect object calls in the client application to specific methods in the server application.

III. Compiling and Linking IDL Source Files

Figure 4 shows how IDL source files are compiled and linked into client and server applications that will utilize the method and apparatus of the present invention. First, an IDL source file 101 is prepared containing IDL interface definitions. An IDL compiler 103 compiles the source file 101. The IDL compiler 103 parses the code 101 to produce an intermediate Pickled IDL file ("PIF") file 105 for storage of the original source file. A code generator 111 then parses the PIF file. The generation of a PIF file is described below in Section VI. Preferably, the IDL compiler and code generator are combined to generate code. The code generator 111 generates files in the language of the client and server applications. If the client and server applications are in different languages, different code generators are used.

Alternatively, the code generator 111 and IDL compiler 103 may be combined in a single application to produce language-specific code. The code generator 111 produces a client stub file 77 containing client stub functions and a server stub file 87 containing definitions for object implementations. The client stub file 77 and the server stub file 87 are compiled by programming language-specific compilers 121, 123 to produce compiled client stub object code and compiled server stub object code. Similarly, a client application 79 and a server application 89 are compiled by programming-language-specific compilers to produce compiled client application object code and compiled server application object code. The client application 79 and the server application 89 also include a header file 119 generated by the code generator 111. The header file 119 contains common definitions and declarations. Finally, a language compiler 121 links the client application object code and the client stub object code to produce an implementation library 71. Similarly, a second language compiler 123 links the server application object code server stub object code to produce another implementation library 81.

In addition, the header file 119, the client stub file 115, and the server stub file 117 include a compact version of each IDL-defined data structure termed Compact IDL Notation ("CIN"). CIN is an ASCII (or other character-based) representation of an IDL data structure utilizing a special notation. The CIN descriptor is contained in the header file 119 which is included by both the client application 123 and the server application 123. The descriptor is also contained in the client and server stub files as well. Since both the client and server applications 121, 123 have access to the CIN, the generic functionality provided by the PCU library 130 can be used by heterogeneous communicants. The creation of CIN by the code generator is described below in Section V.

IV. Transporting Data Structures

Now, the method of the present invention will be described. The method and apparatus of the present invention is implemented by using a group of generic functions or routines 130 that are available to the client and server applications at run-time. These routines are intended to be used in conjunction with data structures originally described in IDL. The principal routines are

PCU_PREPARE, PCU_PACK, and PCU_UNPACK.

An IDL source file may contain numerous type definitions for various data structures. When the source file is compiled and linked into client and server applications, the data structures are used by the client and server applications to perform object calls. A client application may request that an operation be performed on an object using data contained in a particular data structure. The data included in this data structure will be transported to the server application during the object call. The server application must align the data within the data structure properly in order to effectively implement the operation on an object.

The method of the present invention facilitates the transporting of IDL-defined data structures across heterogeneous systems. Figure 5a shows a sample data structure 501 written in IDL. The structure, MyStruct, as written in IDL, includes three components: a char data type component, a long data type component, and a boolean data type component. This type definition is contained in an IDL source file 101, for example, along with interface definitions.

A code generator parses the IDL source file and produces a header file containing a CIN description 502. The CIN descriptor contains a series of ASCII characters that succinctly describes the structure without using identifiers (such as the name of the structure). In this example, the b3 characters identify the data structure as an IDL struct type containing three elements. The C indicates an IDL char type. The F character identifies an IDL long type and the B character identifies a boolean data type.

PCU_PREPARE converts the CIN description of a data type into a "prepared CIN" form which is more convenient to use at run-time than the CIN description. Prior to utilizing the routines PCU_PACK and PCU_UNPACK, the CIN description of each data structure contained in the header file 119, as generated by the code generator 111, must be "prepared". PCU_PREPARE is called once by both the client and the server. Since the call is a relatively expensive one, a single call to PCU_PREPARE during initialization saves valuable system resources. The call is preferably made during an initialization routine of the client and server application. PCU_PREPARE is defined in C as follows:

PCU_PREPARE (

```
        const char    *cinbuf,
        long          cinlen,
        long          prepbufmaxlen,
        void          *prepbuf,
        long          *prepbuf_len,
        long          *cin_used);
```

In this function, *cinbuf* is a pointer to the buffer containing the CIN description of the data structure. The parameter *cinlen* is the size of the CIN. PCU_PREPARE returns a "prepared CIN" that will be stored in the address pointed to by *prepbuf*. To specify a maximum length for *prepbuf*, *prepbufmaxlen* may be set to a particular value. The function also returns *prepbuf_len*, which specifies the size of the prepared CIN contained in *prepbuf*. A value of NULL may be passed as this parameter if this value is not required. The actual number of bytes that were read from *cinbuf* is returned in the parameter **cin_used*. NULL may also be used as this parameter if the value of **cin_used* is not required.

PCU_PREPARE is used to create a prepared CIN, which is a table of *op_tag* data structures that describes the data type, offset, size, and alignment of the CIN-described data structure. PCU_PREPARE creates these *op_tag* structures by initially creating a *ctx* data structure used to pass context to and from each internal function in PCU_PREPARE. Using a *ctx* structure is preferred over passing individual parameters to the various internal functions. The *ctx* structure is defined in C as follows:

```
struct prepare_ctx_tag {
        op_def *op;
        op_def *op_table;
        op_def *op_end;
        const char    *cinptr;
        const char    *cinend;
long        offset;
short       align;
long        size;
long        nr_unbounded;
long        nr_anys;
op_def *prev_branch;
op_def *main_union;
};
```

The fields of the *ctx* structure are as follows. The *op* pointer points to the current operation in the prepared CIN. This pointer is incremented as PCU_PREPARE analyzes each CIN item (as described below). The *op_end* pointer points to the last possible operation in the prepared CIN plus one. The *cinptr* pointer points to the next byte to be read from the entire CIN string being prepared. The *cinend* pointer points to the last byte plus one of the CIN string being prepared. The *offset, align, and size* fields of this *ctx* structure are output parameters of process_cin_item (described below) that specify the offset, required alignment, and size of the processed field in the CIN-described data structure. A running count of the number of unbounded sequences and strings encountered in the processed CIN string is contained in the *nr_unbounded*. A running count of the fields using the IDL "any" data type is contained in the *nr_anys* field. The field *prev_branch* points to a union branch operation previously processed. A list of This field is used to build a list of branch operations whose head is contained in the main union operation. The union operation is pointed to by *main_union*.

Once the *ctx* structure has been created, PCU_PREPARE calls PROCESS_CIN_ITEM for each character in the CIN string, TAKE_LONG for each signed long integer in the CIN string, and TAKE_ULONG for each unsigned long integer in the CIN string. PROCESS_CIN_ITEM processes a single item in the CIN string. The *ctx* structure is passed to PROCESS_CIN_ITEM. PROCESS_CIN_ITEM can be implemented in many ways. Preferably, the function uses a C-language "switch" statement containing a "case" for each possible character in a CIN string. In addition, a case statement may be used to recursively call itself to handle complex structures such as a sequence of struct types or a union of unions.

TAKE_LONG and TAKE_ULONG are used in conjunction with particular data types that are followed by numerals (number of array dimensions, etc...). TAKE_LONG extracts a signed long integer from the CIN buffer and returns the value to PCU_PREPARE. TAKE_ULONG extracts an unsigned long integer from the CIN buffer and returns the value to PCU_PREPARE. These values are used by PCU_PREPARE to create the table of *op_tag* data structures.

For each call, PROCESS_CIN_ITEM modifies the *ctx* data structure. First, PROCESS_CIN_ITEM increments the *op* pointer to ensure that the other fields of the structure correspond to the proper CIN item. In addition, the *size*, *align*,. and *offset* fields of the *ctx* structure are changed. The alignment for each data type is determined based upon the following alignment rules. Base data types are aligned to their size. Thus, a short data type has two-byte alignment, a long has a four-byte alignment, etc... Struct types and union types have the same alignment as the contained field with the highest alignment requirement. Nevertheless, struct and union types, preferably, have an alignment requirement of at least two bytes. Finally, struct and union types are preferably padded to a multiple of their alignment requirement.

When each call to PROCESS_CIN_ITEM returns, PCU_PREPARE creates an *op_tag* data structure based upon the modified *ctx* structure. An array of these *op_tag* structures is then stored in the prepared CIN buffer, prepbuf, after calling PCU_PREPARE. The op_tag structure is a linear structure that can easily be manipulated by other functions. The structure, op_tag, is defined as follows:

```
struct op_tag {
type_def        type;
long            offset;
long            align;
long            size;
long            nr_elements;
long        branch_label;
char        is_default_branch;
char        is_simple;
char        reserve_XXX;
op_def   *sequence_end;
op_def   *next_branch;
op_def   *union_end;
op_def   *default_branch};
```

The *type* parameter indicates the IDL data type of the data structure. The *type_def* type definition is an enumeration of all of the possible data types. The *offset* parameter is the offset of the component data structure from the start of the containing structure or union if the data structure is part of a structure or union. The alignment required by the data type (1, 2, 4, or 8 bytes) is specified by the *align* parameter. The *size* parameter indicates the size of the data structure in bytes including rounding. The *nr_elements* parameter is used for different purposes. For an array, the parameter indicates the total number of elements for all dimensions. For sequences, the parameter indicates the maximum number of occurrences. For strings, the parameter specifies the maximum size excluding zero termination. For structures, it indicates the number of primary fields in the structure. For unions, it indicates the number of fields in the union. The *branch_label* and *is_default_branch* parameters are for union branches only. The *branch_label* parameter contains the case label value that was specified in the IDL specification of the union, while the *is_default_branch* parameter is true if the entry describes the default union branch. The *is_simple parameter* is a boolean value that is true if the data structure is of an IDL base data type and is false if the data structure is a compound type. The **next_branch* parameter is used for unions and union branches and points to the address of the next branch entry belonging to the union. In the case of a union entry, the parameter points to the first branch. For the last branch, the parameter contains the value NULL. The **union_end* parameter points to the address of the next entry following the conclusion of the final branch. The **sequence_end* parameter, used for sequences only, points to the address of the next entry following the sequenced type. The **default_branch* parameter points to the address of the default branch entry. This is used if none of the branches in the branch list matched the union discriminator. If there is no default, the value of *default_branch* is NULL. The *reserve_XXX* parameter allows fields to be added to the op_tag structure without causing errors in existing programs that erroneously assume the size of the prepared CIN.

Figure 5b shows the generated array of *op_tag* structures for the CIN string 502. The first structure 520 specifies the type, offset, size, alignment, and number of members for the MyStruct structure. The next three *op_tag* structures 530, 540, 550 contain the type, offset, size, and alignment for each field in the MyStruct structure. This array of structures is stored in a buffer, *prepbuf*, that will be used by PCU_PACK and PCU_UNPACK to send structured data across a file or to a network.

Once the data structure has been "prepared" and the array of *op_tag* structures is stored in *prepbuf*, various messages stored in that data structure can be packed into a buffer and transported using PCU_PACK. PCU_PACK is used to copy a structured data type into an output buffer during transport to a file or across the network. PCU_PACK supports all IDL constructs including unions, unbounded sequences/strings and "any" types.

PCU_PACK stores the components of a structured data type into an output buffer based upon a specified format. PCU_PACK is defined in C as follows:

```
PCU_PACK (
        char            dst_integer_fmt,
        char            dst_real_fmt,
        char            dst_char_fmt,
        const void      *prepbuf,
        const void      *inbuf,
        long            outbuf_max_len,
        void            *outbuf,
        long            *outbuf_len);
```

The first three parameters specify how data is to be packed into the output buffer. These parameters may be caller-defined functions for performing the conversion as provided by the caller. The first parameter, dst_integer_fmt, specifies the format to be used for short, long and long data types in the output buffer. Examples of possible values for this format are PCU_INTEGER_BIGENDIAN which specifies an integer representation where the byte significance decreases with increasing address or PCU_INTEGER_LITTLEENDIAN which specifies an integer representation where the byte significance increases with increasing address. The parameter dst_real_fmt specifies the format to be used for float and double data types in the output buffer. Sample values for this parameter are PCU_REAL_IEEE which specifies a floating point number representation using the standard IEEE format or a vendor-specific value, such as PCU_REAL_T16 which specifies a floating point number representation using the Tandem T16 format, for example. The third parameter, dst_char_fmt specifies the format to be used for char and string types in the output buffer. One possible value for this parameter is a character representation using the ISO Latin-1 format, a super-set of ASCII. Anther possible value is EBCDIC, which permits compatibility with IBM hosts.

The *prepbuf parameter, as stated above, is a pointer to the address containing the prepared CIN description as returned by PCU_PREPARE. The *inbuf parameter is a pointer to the address of the structured data to be stored into the output buffer. The *outbuf parameter is a pointer to the address of the output buffer that receives the actual packed data. The maximum number of bytes that can be accommodated by outbuf is contained in the outbuf_max len parameter. The number of bytes actually written to outbuf is returned by the outbuf_len parameter. A value of NULL may be passed as this parameter if the number of bytes is not needed. If PCU_SHORTOUTBUF is returned by the function, then the outbuf_len parameter gets the required outbuf size.

Accordingly, to dynamically allocate memory for the output buffer, the client application can call PCU_PACK twice. On the first call, outbuf_max_len is set to zero. PCU_PACK will then return PCU_SHORTBUF and outbuf_len will contain the required output buffer size. The correct amount of memory for the output buffer can then be allocated prior to calling

PCU_PACK for a second time.

PCU_PACK initially creates a *ctx* structure. This *ctx* structure provides a similar function as the context structured used by PROCESS_CIN_ITEM. The structure allows large amounts of context to be shared between PCU_PACK and the lower-level routines that are called by PCU_PACK_ENGINE. This *ctx* structure is used by the underlying functions to PCU_PACK and is defined as follows:

```
struct pack_ctx_tag {
          char          dst_integer_fmt,
          char          dst_real_fmt,
          char          dst_char_fmt,
          char          *outptr,
          char          *outbuf_end

}
```

The requested destination format as specified in the call to PCU_PACk are passed to the *ctx* structure. These three fields are needed in case PCU_PACK must be called recursively to handle an IDL "any" type. The *outptr* pointer points to the next byte to be written into the output buffer. Even if the output buffer is full, the pointer continues to be updated. This allows the correct size to be returned to the caller in case of overflow. The caller can then adjust the size of the output buffer. The pointer *outbuf_end* points to the last byte plus one in the output buffer.

PCU_PACK calls an internal function, PCU_PACK_ENGINE. PCU_PACK_ENGINE receives pointers to lower-level functions that perform the actual packing of data into the output buffer. PCU_PACK also receives a pointer to *prepbuf*, a pointer to the data to be packed (contained in *inbuf*), and a pointer to the *ctx* structure created by PCU_PACK. PCU_PACK_ENGINE goes element-by-element through the *prepbuf* buffer and calls the appropriate lower-level function for the element based upon the *type* of the element (as specified by the type contained in the *op_tag* structure) and based upon the dst_XXX_fmt parameter to PCU_PACK. PCU_PACK_ENGINE provides the address to the input buffer containing the structured data, the data type (via a CIN character), the *ctx* structure address, and the size of the data in the input buffer to pack into the output buffer (as specified by the *size* field of the *op_tag* structure).

PCU_PACK_ENGINE calls the appropriate lower-level function based upon the type of data contained in the *op_tag* data structure and the conversion specified on the call to PCU_PACK. The lower-level functions are known, lower-level functions that pack data either transparently or perform some specified conversion (BIGendian to LITTLEendian, e.g.). Each caller-supplied function takes data from the input buffer and places it into an output buffer. The number of bytes to be taken from the input buffer is specified by the size parameter provided to the function from PCU_PACK_ENGINE. Once the data has been placed in the output buffer, the lower-level function modifies the *outptr* parameter of the *ctx* structure to point to the byte following the last byte written to the output buffer.

PCU_PACK_ENGINE uses the various lower-level functions to store data in the output buffer as follows. The structured data types in the input buffer are stored densely (byte-aligned) in the output buffer in the same order as they were originally defined in IDL. The contents of any padding fields inserted by the code generator to achieve correct alignment are discarded. Similarly, the functions do not place default values in those fields.

Base type data structures are stored in the output buffer in the representation specified by the *dst_XXX_fmt* parameters on the call to PCU_PACK. Typically, these parameters are set to the packer's native format without any conversion. Thus, the server application (the unpacker) would perform the actual conversion. The routines utilized in the present invention, however, permit the packer to perform a conversion of the data structures as well.

The representation of shorts, unsigned shorts, longs, unsigned longs, long longs, and unsigned long longs are specified in the *dst_real_fmt* parameter to PCU_PACK. This parameter specifies the format for representing floating point numbers. The alignment of floats and doubles are specified by the *dst_real_fmt* parameter. This parameter corresponds to the format for representing integers. The representation of chars are specified by the *dst_char_fmt* parameter. The dst_char_fmt parameter specifies a format for representing characters. Booleans and octets are not realigned. The "any" type is stored as an unsigned long specifying the length of the CIN description (whose alignment is based upon the dst_integer_fmt parameter), a CIN string describing the type (an unconverted ASCII string), and the data itself (stored based upon these conversion rules).

Compound types such as arrays and unions are also realigned. Arrays are stored with no padding between elements. Sequences are stored as unsigned long integers indicating the number of occurrences followed by that number of occurrences. Any padding between occurrences is removed. The format of the long integers depends upon the *dst_integer_fmt* parameter as stated above. A string is stored as an unsigned long indicating the length of the string followed by that particular number of characters stored as chars. The format of the chars is determined by the *dst_char_fmt* parameter. Structures are stored field by field without padding. Unions are stored as a long followed by the active union branch.

On the receiving end, the server application must extract the unstructured data type and its appendages from the buffer that was packed using PCU_PACK. PCU_UNPACK then places this unstructured data into a data structure based upon the prepared CIN for the data structure. PCU_UNPACK is defined as follows:

```
    PCU_UNPACK (
            char          src_integer_fmt,
            char          src_real_fmt,
            char          src_char_fmt,
            const void          *prepbuf,
            const void          *inbuf,
            long          inbuf_len,
            long          outbuf_max_len,
            void          *outbuf,
            long          *outbuf_len,
        long            *inbuf_used);
```

The first three parameters correspond to the first three parameters of PCU_UNPACK. These parameters specify the format of data types as stored in the input buffer. These parameters are preferably identical to their PCU_PACK counterparts. The *prepbuf parameter is a pointer to the address containing the prepared CIN description as returned by PCU_PREPARE. The address of the input buffer is pointed to by *inbuf. The length of inbuf is specified by inbuf_len. The address of the output buffer is pointed to by *outbuf. The maximum number of bytes that can be accommodated by outbuf is specified by outbuf_max_len. The parameter *outbuf_len obtains the number of bytes actually written to outbuf. The number of bytes read from the input buffer is specified by *inbuf_used. If the number of written bytes or the number of read bytes are not needed, NULL may be passed as the value for these parameters.

PCU_UNPACK creates a *ctx* structure that is used to pass context around to the internal functions of PCU_PACK. This *ctx* structure is used by the underlying functions to PCU_UNPACK and is defined as follows:

```
    struct pack_ctx_tag {
            char        src_integer_fmt,
            char        src_real_fmt,
            char        src_char_fmt,
            char        *inptr,
            char        *inbuf_end

    }
```

The first three parameters are the formats passed to PCU_UNPACK. These parameters are needed by the internal functions in case PCU_PACK is called recursively to handle an IDL "any" type. The *inptr* pointer points to the next byte to be read from the input buffer. The *inbuf_end* pointer points to the last byte plus one in the input buffer.

After creating the context structure, PCU_UNPACK calls PCU_UNPACK_ENGINE which provides the functionality for PCU_UNPACK. PCU_UNPACK_ENGINE receives pointers to caller-supplied function for extracting data from the input buffer (the output buffer provided by PCU_PACK) and placing it in an output buffer. The prepared CIN buffer is also provided as a parameter. PCU_UNPACK_ENGINE goes element-by-element through the prepared CIN and stores the data into a data structure as specified by the *offset* and *size* fields of the *op_tag* structures.

For each element in the prepared CIN buffer, PCU_UNPACK_ENGINE calls the appropriate lower-level user-specified function to perform the unpacking and converting. PCU_PACK_ENGINE passes the data type and size of the data to be read from the input buffer along with the address of the output buffer to write the data. PCU_UNPACK_ENGINE also passes the *ctx* data structure to each of the functions. Each caller-specified function then extracts the data from the input buffer and places the data into a data structure. The number of bytes to be written from the input buffer to the output buffer is determined by the size parameter. For compound types, PCU_UNPACK_ENGINE provides additional parameters to the caller-supplied functions. If the data structure is an array, the number of elements in the array is provided. If the data structure is a sequence, PCU_UNPACK_ENGINE provides the maximum number of elements in the sequence along with the actual number of elements. If the data structure is a string, the maximum size and actual size of the string are provided to the caller-supplied functions. If the compound type is a struct data type, the number of members of the structure are provided.

Now, with reference to Figures 6 and 7, the method of the present invention will be described. Figure 6 is a flow chart of the client side of the method of the present invention. Prior to performing the method of the present invention, as stated above, compact descriptions of data structures are created by the code generator 111 and are included in the client and server stubs. This description can be created using the method described in Application No. XXX. The client and server stubs are compiled and linked into the client and server applications. Once the client stubs have been linked into the client application, in a first step 601, the client application creates a prepared CIN description by calling the function PCU_PREPARE. PCU_PREPARE takes the CIN description of the data structure and, in step 603, converts the CIN to an array of op_tag data structures by calling PROCESS_CIN_ITEM for each element of the CIN description. Each structure contains information regarding the type, offset, alignment, and size of the CIN-described data structure. A table of these structures are then stored in a memory buffer called prepbuf, in step 605.

In step 607, the client application calls PCU_PACK which packs the data structure by copying the data into an output buffer based upon the size as specified by the size field of each *op_tag*. PCU_PACK removes any alignment padding fields from the data structure and places the data structure, in step 609, into an output buffer. Once the data structure has been packed into the output buffer, the data is transported in step 611. The data structure may be transported across a wire to a server application or transported to a file, such as a disk file. If another request involving the same data structure is made, this request is packed and the client application repeats steps 605-611 for the new request. The CIN description of the data structure need not be "prepared" again.

Figure 7 shows the server side of the method of the present invention. The server application, in step 701, calls PCU_PREPARE to obtain a prepared description of the CIN. The "prepare" step is similar to step 601 described above. The server then calls PCU_UNPACK in step 703 to extract a structured data type and all of its appendages from the buffer that was packed using PCU_PACK. In step 705, the structure is unpacked based upon the parameters passed to the function (the same parameters passed to the PCU_PACK function). While extracting the data structure, the structure is realigned in step 707 from the format specified in the input buffer of the server to the native alignment of the server. If another request arrives at the server, the server can call PCU_UNPACK to unpack the request without preparing the data structure.

V. Compact IDL Notation

Figure 8 is a flow chart depicting the generation of a CIN descriptor from an IDL data type, operation, and interface contained in an IDL source file. It will be understood that the steps of Figs. 8-10 are implemented by a CPU of a data processing system executing computer instructions stored in memory. In step 801, a code generator begins with the first line of an IDL source file and determines the data structure, interfaces, or operation described in the source file. If the described data structure is an interface, the code generator follows the directions shown in Figure 10. If the described data structure is an operation within an interface, the generator follows the directions in Figure 9. If the data structure is a data type (or a parameter to an operation as discussed below), the generator generates a single character based upon a table of definitions. It should be noted that different characters may be used than those shown in the charts contained herein. Each chart contains only a preferred ASCII character. Chart A shows a preferred definition table that includes the character strings used to denote the various IDL base types.

IDL Base Type	**Representation**
Any	A
Boolean	B
Char	C
Double	D
Float	E
Long	F
Long Long	G
Octet	H
Short	I

Unsigned Short	J
Unsigned Long	K
Unsigned Long Long	L
Void	M

Chart A

As shown in Chart A, simple character strings are used to represent base types in a CIN descriptor.

If the data type is a compound type, such as an array or structure (struct type), a series of different steps are followed. In step 811, a character is generated that indicates the start of the compound type. Chart B shows a sample table that includes the character strings used to denote the start of various IDL compound types.

IDL Compound Type	**Representation**
Array	a
Struct	b
Sequence	c
String	d
Union	e
Union Branch	f

Chart B

The particular representation of each compound type is handled differently according to type.

IDL defines multidimensional, fixed-size arrays for each base type. The array size is fixed at compile time. Arrays are represented in CIN as follows:

a *nr_dimensions size_1. [size_2 , . . . size_n] base type*

The generation of characters for the array is shown in Steps 813, 815, and 817. In this array representation, the character a represents the start of the array (as shown in Chart B). The character *nr_dimensions* is a numeral indicating the number of dimensions in the array. The characters *size_1*, *size_2*, *size_n* indicate the size of the array in the first, second, and nth dimension of the array, respectively. For each element of the array, *base-type* is the descriptor for each element. The representation for the various base types is derived from the original base type table shown in Chart A.

IDL defines user-defined struct types. Each struct is composed of one or more fields of base or compound data types. Structs are represented in CIN as follows:

b *nr_fields field_1 [field_2 . . . field_n]*

The generation of characters to describe a struct is shown in steps 819 and 821. As shown in Chart B, the character **b** indicates the start of the struct. The numeral *nr_fields* indicates the number of fields in the struct. The fields in the struct are described by the descriptors *field_1*, *field_2*, *field_n*. Each field is a base or compound type. Base type fields of the struct are described as shown in Chart A. Compound types are described as shown herein (i.e., arrays are described with the start character **a** along with the number of dimensions and the size of each dimension, etc...).

IDL defines sequences of data types. A sequence is a one-dimensional array with two characteristics: a maximum size (fixed at compile time) and a length (determined at run time). Sequences are represented as:

c *nr_occurrences base_type*

The generation of characters for a sequence is shown in steps 823 and 825. In this representation, **c** indicates the start of the sequence as shown in Chart B. The character *nr_occurrences* specifies how many occurrences of the data type are included in the sequence. The number of occurrences is then followed by the actual descriptor, *base_type*, for each occurrence of the data type in the sequence. If the data type is a base type, the appropriate descriptor from Chart A is used. If the sequence consists of compound types, the descriptors are created as described herein. A sequence of sequences is possible.

IDL defines the string type consisting of all possible 8-bit quantities except null. A string is similar to a sequence of chars. Strings are represented in CIN as:

d *size*

The start of the string is indicated by the **d** character. The size of the string is represented by the *size* character generated in step 827.

In IDL, unions are a cross between the "union" of the C programming language and the C "switch" statement. In other words, the union syntax includes a "switch" statement along with a "case" label indicating the union branches. IDL unions must be discriminated; that is, the union header must specify a typed tag field that determines which union member to use for the current instance of a call. Unions and union branches are represented in CIN as follows:

e *nr_fields* **f** *label_1 field_1 [***f** *label_2 field_2 . . . label_n field_n]*

The generation of characters to represent unions and union branches is shown in steps 829, 831, 833 and 835. In this representation, **e** indicates the start of a union and **f** indicates the start of a union branch within the union. The number of fields in the union is specified by the character *nr_fields*. The case label value for each field is indicated by the character *label_1*. If the field is a default, the label is omitted. The descriptor for each field, *field_1*, then follows. The union fields may be either a base or a compound type. Accordingly, the field descriptor for a base type may be generated based upon Chart A. Compound types are generated as described herein.

As seen from the descriptors for base and compound types, the CIN description does not include the identifiers contained in the original IDL source file and a generic descriptor is generated. Thus, even the most complex data structures can easily be represented in string format. The following is a sample structure originally described in IDL:

```
union coordinate_def switch (boolean) {
        case FALSE;
                struct cartesian_def {
                        long x;
                        long y;
                } cartesian;
        case TRUE;
```

```
                struct polar_def {
                        unsigned long radius;
                        unsigned long theta;
                } polar;
        };
        typedef sequence<coordinate_def, 100> coordinate_list_def;
```

Using the method of the present invention, the above-described data structure would be represented in CIN as:

"c100+e2+f0+b2+FFf1+b2+KK"

In a preferred embodiment, positive numerals are followed by a plus sign ("+"). Negative numbers are terminated by a negative sign ("-"). While negative numbers may not occur frequently, their use may be required for certain data types, such as union case labels (i.e., the case discriminator may be a negative number). The CIN descriptor shown above is explained as follows: A data structure consisting of a sequence (c) with a maximum 100 elements (100+), each element consisting of a union (e) with two fields (2+). If the discriminator is zero (FALSE) (f0+) then one variant is a struct (b) containing two fields (2). The first field is a signed long (F). The second field is a signed long (F). If the discriminator is 1 (TRUE) then (f1+) the second variant is a struct containing (b) two fields (2+). The first field is an unsigned long (K). The second field is an unsigned long (K).

If an operation is to be described in CIN, then the method continues at step 951. Figure 9 is a flow chart depicting the steps followed in generating a descriptor for an operation. An operation descriptor is generated in CIN as:

operation_synopsis
operation_id
operation_attribute
nr_params
param_1, param_2 . . . param_n,

nr_exceptions

exception_1, exception_2 . . . exception_n

nr_contexts

context_1, context_2 . . . context_n

In step 951, the code generator generates a unique integer, *operation_synopsis*, that is derived from the string constituting the remainder of the operation's descriptor. The integer is derived by performing, for example, a cyclic redundancy check on the remaining characters in the CIN descriptor. Next, the code generator generates a unique string, *operation_id*, derived from the original IDL name of the operation. Next, in step 955, the code generator generates *operation_attribute*, a character that indicates the attributes (none or "oneway") of the operation. For instance, if the operation has no oneway attribute, the character **A** is generated. If, however, the operation's attribute is oneway, the code generator generates the character **B**. The character *nr_params* is an integer that indicates how many parameters are included in the operation. If the operation has a non-void return type then the first parameter is the result. The *param_1* descriptor includes a character that indicates the direction of the parameter (in, out, inout, or function result) followed by the actual parameter data type. The code generator, for example, generates the characters **A**, **B**, **C**, and **D** for the directions of in, out, inout, and function result, respectively. For the specific parameters, the method returns to step 807 in Figure 8. When the data type of each parameter has been described, the number of exceptions is identified by the integer *nr_exceptions*. The structure description for each exception is then described by returning to step 819, which describes structures. The integer *nr_contexts* indicates the number of context names held by the operation. The names are then generated in strings, *context_1, context_2, context_n*.

The following are two sample operations originally described in IDL:

```
interface Math {
        long Add (in long x, in long y);
        long Subtract (in long x, in long y);
```

};

Using the method of the present invention, the above-described Add operation would be represented in CIN as:

126861413+3+ADDA3+DFAFAF0+0+

The CIN descriptor for the operation is described as follows. The beginning numeral (126861413) is derived from the remainder of the CIN by performing a cyclic redundancy check on the string "3+ADDA3+DFAFAF0+0+". The operation id contains three characters (3+). Those three characters are the string "ADD"–the operation id. All IDL identifiers must be unique and independent of case. Thus, operation id's are capitalized. The operation does not include the oneway attribute (A). The operation includes three "parameters" (3+). Since the function returns a result, the first "parameter" is actually a function result (D). The function result is a signed long (F). The next parameter (actually the first parameter) is an in parameter (A). The parameter is of typed signed long (F). The third parameter is an in parameter (A) of typed signed long (F). There are no exceptions (0+) and no contexts (0+).

Similarly, the CIN descriptor for the Subtract operation would be:

453399302-9+SUBTRACTA3+DFAF0+0+

Interfaces are similarly described using the method of the present invention. Figure 10 shows the generation of interface descriptors. Interfaces are defined as follows:

nr_operations

operation_spec_1

operation_spec_2

operation_spec_n

The integer, *nr_operations* indicates how many operations are contained in the interface. Each operation is then described in *operation_spec_1, operation_spec_2, operation_spec_n* according to the above-described method for generating an operation descriptor. The code generator 112, thus goes to step 951 in Figure 9 to describe each operation.

Using the method of the present invention, the above-described Math interface would be represented in CIN as:

2+126861413+3+ADDA3+DFAFAF0+0+453399302-9+SUBTRACTA3+DFAF0+0+

The interface includes two operations (2+). The operation descriptors for the Add and Subtract operations follow the character indicating the number of operations.

As stated above, the CIN descriptors are contained in a header file that is linked into both the client and server applications. Thus, both the client and the server can make use of the descriptor as each sees fit. The CIN may be used in many ways. For example, a CIN description of a data type may be useful in creating generic functions to pack and unpack structured data types.

CIN descriptions may also be used to compare interfaces quickly. For example, a server application may have a header file containing two interfaces described in CIN. The server may then compare the ASCII string descriptions using known string comparison functions. If the server determines that the CIN descriptions are identical (or similar), the server may implement the operations of both interfaces using common methods in the server application. Thus, the CIN can be used to save time coding multiple methods for different (but similar) interfaces.

VI. Creating a Pickled IDL Format Data Structure

The Pickled IDL Format ("PIF") data structure is designed to be used in conjunction with IDL compilers and code generators loaded in the client memory 23 and server memory 17. The data structure is based upon an IDL source file stored in memory 23 or in memory 17. The source file may also be contained on a computer-readable medium, such as a disk. The data structure of the present structure contains a parse tree representing the IDL source file. The data structure can be stored in memory 23 or in memory 17 or on a computer-readable medium, such as a disk. The data structure that represents the source file is referred to as a Pickled IDL Format ("PIF"). The PIF file can be accessed at run-time by clients and servers that use the interfaces defined in the source file. The parse tree contained in the PIF file is an array using array indices rather than pointers. The use of array indices permits the resulting parse tree to be language-independent. The first element of the array is unused. The second element of the array (index 1) is the root of the parse tree that acts as an entry point to the rest of the parse tree.

The data structure, *tu* 1101, is shown in Figure 11, and defined in IDL as follows:

```
struct tu_def {
          sequence<entry_def>          entry;
          sequence<string>             source;
}
```

The data structure 1101 contains a sequence (a variable-sized array) of parse tree nodes 1105, each of type entry_def (defined below) and a sequence of source file lines 1107. The sequence of source file lines 1107 is a sequence of strings containing the actual source code lines from the IDL source file.

Each parse tree node (or "entry") 1105 consists of a fixed part containing the name of the node and its properties as well as a variable portion that depends upon the node's type. The parse tree node is shown in Figure 12 and defined in IDL as follows:

```
struct entry_def {
          unsigned long  entry_index;
          string                 name;
          string                 file_name;
          unsigned long  line_nr;
          boolean                in_main_file;
          union u_tag switch (entry_type_def) {
                    case entry_argument: argument_def argument_entry;
                    case entry_array: array_def array_entry;
                    case entry_attr: attr_def attr_entry;
                    case entry_const:  const_def const_entry;
                    case entry_enum: enum_def enum_entry;
```

```
                case entry_enum_val: enum_val_def enum_val_entry;
                case entry_except: except_def except_def_entry;
                case entry_field: field_def field_def entry;
                case entry_interface: interface_def interface_entry;
                case entry_interface_fwd: interface_fwd_def interface_fwd_entry;
                case entry_module: module_def module_entry;
                case entry_op: op_def op_entry;
                case entry_pre_defined: pre_defined_def pre_defined_entry;
                case entry_sequence: sequence_def sequence_entry;
                case entry_string: string_def string_entry;
                case entry_struct: struct_def struct_entry;
                case entry_typedef: typedef_def typedef_entry;
                case entry_union: union_def union_entry;
                case entry_union_branch: union_branch_def union_branch_entry;
        } u;
    };
```

The fixed part of the parse tree node includes *entry_index* 1205, an unsigned long which is the index for this particular entry in the parse tree. The unqualified name of the entry is contained in the field *name* 1207. The name of the original IDL source file is contained in the field *file_name* 1211. The field line_nr 1213 contains the line number in the IDL source file that caused this parse tree node to be created. The boolean *in_main_file* 1215 indicates whether or not the entry is made in the IDL source file specified on the command line or whether the entry is part of an "include" file. Following these fields, the parse tree node includes a variable portion--a union 1217 having a discriminator, *entry_type_def*. The union discriminator, *entry_type_def*, specifies the type of node and which variant within *entry_def* is active. *Entry_type_def* is an enumeration defined as follows:

```
    enum entry_type_def {
            entry_unused,
            entry_module,
            entry_interface,
            entry_interface_Fwd,
            entry_const,
            entry_except,
            entry_attr,
            entry_op,
            entry_argument,
            entry_union,
            entry_union_branch,
            entry_struct,
            entry_field,
            entry_enum,
            entry_enum_val,
            entry_string,
            entry_array,
            entry_sequence,
            entry_typedef,
            entry_pre_defined
    };
```

Entry_type_def includes a list of the various types of parse tree entries. Each parse tree entry represents a constant integer that is used in the switch statement contained in *entry_def*. For each entry, the union *u_tag* will include a different type of structure. The first enumerated value entry_unused corresponds to the value zero and is not used in determining the type of the union.

If the parse tree entry is a module (specified by the value entry_module) the variable portion of the parse tree entry is a data structure including a sequence of module definitions.

Each module definition is an unsigned long acting as an index in the parse tree array.

If the parse tree entry is an interface, as specified by the value entry_interface, the variable portion of the parse tree is a data structure including a sequence of local definitions and a sequence of base interfaces from which this interface inherits. If the parse tree entry is a forward declaration of an interface (entry_interface_fwd), the union is an unsigned long containing the index of the full definition.

Constants (entry_const) are represented in a parse tree node as a structure containing the value of the constant. A union and switch/case statement are preferably used to discriminate between the various base type constants (boolean constant, char constant, double constant, etc...) that may be included in the source file.

Exceptions (entry_except) are represented in a parse tree node as a structure containing a sequence of fields. An attributes (entry_attr) is represented as a data structure containing a boolean value that indicates whether the attribute is read-only and an unsigned long that indicates the data type.

If the parse tree entry is an operation (op_def), the variable portion 1217 of the entry data structure 1105 is a data structure as shown in Figure 13. The data structure 1217 contains a boolean 1305 that indicates whether or not the operation has a one-way attribute, an unsigned long 1307 that indicates the return type, a sequence of arguments 1309 to the operation, a sequence of exceptions 1311 to the operation, and a sequence of strings 1313 that specify any context included in the operation. If the parse tree entry is an argument to a particular operation (entry_argument), the variable portion of the parse tree entry is a structure containing unsigned longs that indicate the data type and direction of the argument.

If the parse tree entry is a union (entry_union), it is represented in the parse tree entry as shown in Figure 14. The data structure 1217 contains an unsigned long specifying the discriminator 1403 and an unsigned long specifying the type 1405. The type is preferably specified using an enumerated list of base types. The structure 1217 further includes a sequence of the union's fields 1407. If the parse tree entry is a union branch (entry_branch), the variable portion of the parse tree entry is a structure containing an unsigned long indicating the base type of the branch, a boolean indicating whether or not the branch includes a case label, and the value of the discriminator. Since the value is of a particular data type, preferably an enumerated list of the various base types is used to specify the value within the structure used to represent the union branch.

For data structures (entry_struct), the variable portion of the parse tree entry includes a structure containing a sequence of the specified structure's fields. Enumerated values (entry_enum) are represented by a structure containing a sequence of enumerated values. Enumerations of an enumerated type (entry_enum_val) are represented in the parse tree entry by a structure containing an unsigned long holding the enumeration's numerical value.

If the parse tree entry is a string (entry_string), the variable portion of the parse tree entry is a structure containing the string's maximum size. A maximum size of zero implies an unbounded string. An array (entry_array) is represented in the parse tree entry by a structure containing an unsigned long holding the array's base type and a sequence of longs holding the array's dimensions. A sequence (entry_sequence) is represented by a structure containing unsigned longs holding the sequence's base type and the sequence's maximum size.

For type definitions (entry_typedef), the parse tree entry includes a structure containing an unsigned long value indicating the type definition's base type. Predefined types (entry_pre_defined) are represented by a structure containing the data type. To specify the type, preferably an enumeration of the various base types are used.

Once the IDL source file has been described using the *tu* data structure, the data structure may be transported to a file or database using any known methods.

Having thus described a preferred embodiment of a method and apparatus for converting IDL-defined data structures to and from a format convenient for transport, it should be apparent to those skilled in the art that certain advantages of the within system have been achieved. It should also be appreciated that various modifications, adaptations, and alternative embodiments thereof may be made within the scope and spirit of the present invention. For example, IDL-defined data structures have been illustrated, but it should be apparent that the inventive concepts described above would be equally applicable to hand-written data structures if the structures follow the same alignment rules as the code generators. The invention is further defined by the following claims.

CLAIMS

What is Claimed is:

1. A method for transporting data from a first computer memory to at least one of a data file and a second computer memory, the data being stored in a first data structure having at least one field, the method comprising the steps of:

generating a string description of the first data structure;

storing the string description of the first data structure in the first computer memory and the second computer memory;

generating a second data structure containing a size, alignment, and type of the at least one field;

storing the second data structure in the first computer memory and the second computer memory;

storing the data in a buffer of the first computer memory based upon the string description of the first data structure;

transporting the buffer to the at least one of the data file and the second computer memory;

extracting the data from the buffer; and

storing the data in a third data structure based upon the string description of the data structure.

2. The method for transporting a data structure, as recited in Claim 1, wherein the step of storing the data in a buffer further comprises the steps of:

storing floating point and double type components of the data structure in a format for representing floating point numbers;

storing long, long long, unsigned long, unsigned long long, short, and unsigned short type components of the data structure in a format for representing integers; and storing character type components of the data structure in a format for representing characters.

3. The method for transporting a data structure, as recited in claim 2, wherein the step of storing the data structure further comprises the steps of:

converting "any" type components of the data structure to an unsigned long that specifies a length of the string description, the string description, and components of the "any" data structure; and

storing the unsigned long in the format for representing integers;

storing the string as an unsigned long indicating the length of the string followed by a plurality of characters;

storing each of the plurality of characters in the format for representing characters; and

storing each component of the "any" data structure according to a predetermined format.

4. The method for transporting data structures, as recited in claim 2, wherein the step of storing the at least one data structure based upon a predetermined format further comprises the steps of:

storing elements of an array component of the data structure in an unpadded format;

storing a sequence component of the data structure in the format for representing integers; and

converting a string component of the data structure to an unsigned long type indicating a length of the string, followed by a plurality of characters;

storing the unsigned long representing the string component of the data structure in the format for representing integers; and

storing each of the plurality of characters in the format for representing characters.

5. The method for transporting data structure, as recited in claim 2, wherein the format for representing floating point numbers is IEEE format.

6. The method for transporting data structure, as recited in claim 2, wherein the format for representing floating point numbers is Tandem T16 format.

7. The method for transporting data structure, as recited in claim 2, wherein the format for representing integers specifies that byte significance of an integer decrease with increasing address.

8. The method for transporting a data structure, as recited in claim 2, wherein the format for representing integers specifies that byte significance of an integer increase with increasing address.

9. The method for transporting a data structure, as recited in claim 2, wherein the format for representing characters is ISO Latin-1.

10. The method for transporting a data structure, as recited in claim 1, further comprising the step of:

obtaining information about the at least one data structure; and

allocating memory in the output buffer of the first computer.

11. A method for transporting an Interface Definition Language-defined aligned data structure from a first computer memory having a description of the data structure in a string format to at least one of a data file and a second computer memory having the string description of the data structure, the method comprising the steps of:

converting the string description to a format containing information about the alignment and size of the data structure;

removing at least one alignment field from the data structure;

storing floating point and double type components of the data structure in a buffer of the first computer, the floating point and double type components of the data structures being stored in a format for representing floating point numbers ;

storing long, long long, unsigned long, unsigned long long, short, and unsigned short type components of the data structure in the buffer of the first computer, the long, long long, unsigned long, unsigned long long, short, and unsigned short type components of the data structure being stored in a format for representing integers;

storing character type components of the data structure in the buffer of the first computer, the character type components of the data structures being stored in a format for representing characters;

transporting the buffer to the at least one of a data file and the second computer memory;

extracting the data structure from the buffer based upon the format for storing each component of the data structure; and

12. The method for transporting a data structure, as recited in claim 11, wherein the format for representing floating point numbers is IEEE format.

13. The method for transporting a data structure, as recited in claim 11, wherein the format for representing floating point numbers is Tandem T16 format.

14. The method for transporting a data structure, as recited in claim 11, wherein the format for representing integers specifies that byte significance of an integer decrease with increasing address.

15. The method for transporting a data structure, as recited in claim 11, wherein the format for representing integers specifies that byte significance of an integer increase with increasing address.

16. The method for transporting a data structures, as recited in claim 11, wherein the format for representing characters is ISO Latin-1.

17. A media readable by at least one machine having a description of an Interface Definition Language-defined aligned data structure in string format, embodying instructions for causing the at least one machine to perform a method of transporting the data structure from a memory of the at least one machine to at least one of a data file and a second machine memory having the ASCII description of the data structure, the method comprising the steps of:

converting the string description to a format containing information about the alignment and size of the data structure;

removing at least one alignment field from the data structure;

storing the data structure in a buffer in the memory of the first machine, the at least one data structure being stored in a predetermined format and based upon the string description;

transporting the buffer from the memory of the first machine to the at least one of a data file and a second machine;

extracting the data structure from the buffer, the data structure being extracted according to the predetermined format; and

aligning the at least one data structure.

18. The media, as recited in claim 17, wherein the step of storing the at least one data structure based upon a predetermined format further comprises the steps of:

storing floating point and double type component of the data structure in a format for representing floating point numbers;

storing long, long long, unsigned long, unsigned long long, short, and unsigned short type components of the data structure in a format for representing integers; and

storing character type components of the data structures in a format for representing characters.

19. The media, as recited in claim 17, wherein the step of storing the at least one data structure based upon a predetermined format further comprises the steps of:

converting "any" type components of the data structures to an unsigned long specifying a length of the original ASCII string, the ASCII string, and data structures within the "any" data structure; and

storing the unsigned long in the format for representing integers;

storing the ASCII string as an unsigned long indicating the length of the string followed by a plurality of characters;

storing each of the plurality of characters in the format for representing characters; and

storing each data structure within the "any" data structure in a predetermined format.

20. The media, as recited in claim 17, wherein the step of storing the at least one data structure based upon a predetermined format further comprises the steps of:

storing elements of an array component of the data structure in an unpadded format;

storing a sequence component of the data structure in the format for representing integers; and

representing a string component of the data structure as an unsigned long type indicating a length of the string, followed by a plurality of characters;

storing the unsigned long representing a string in the format for representing integers; and

storing each of the plurality of characters in the format for representing characters.

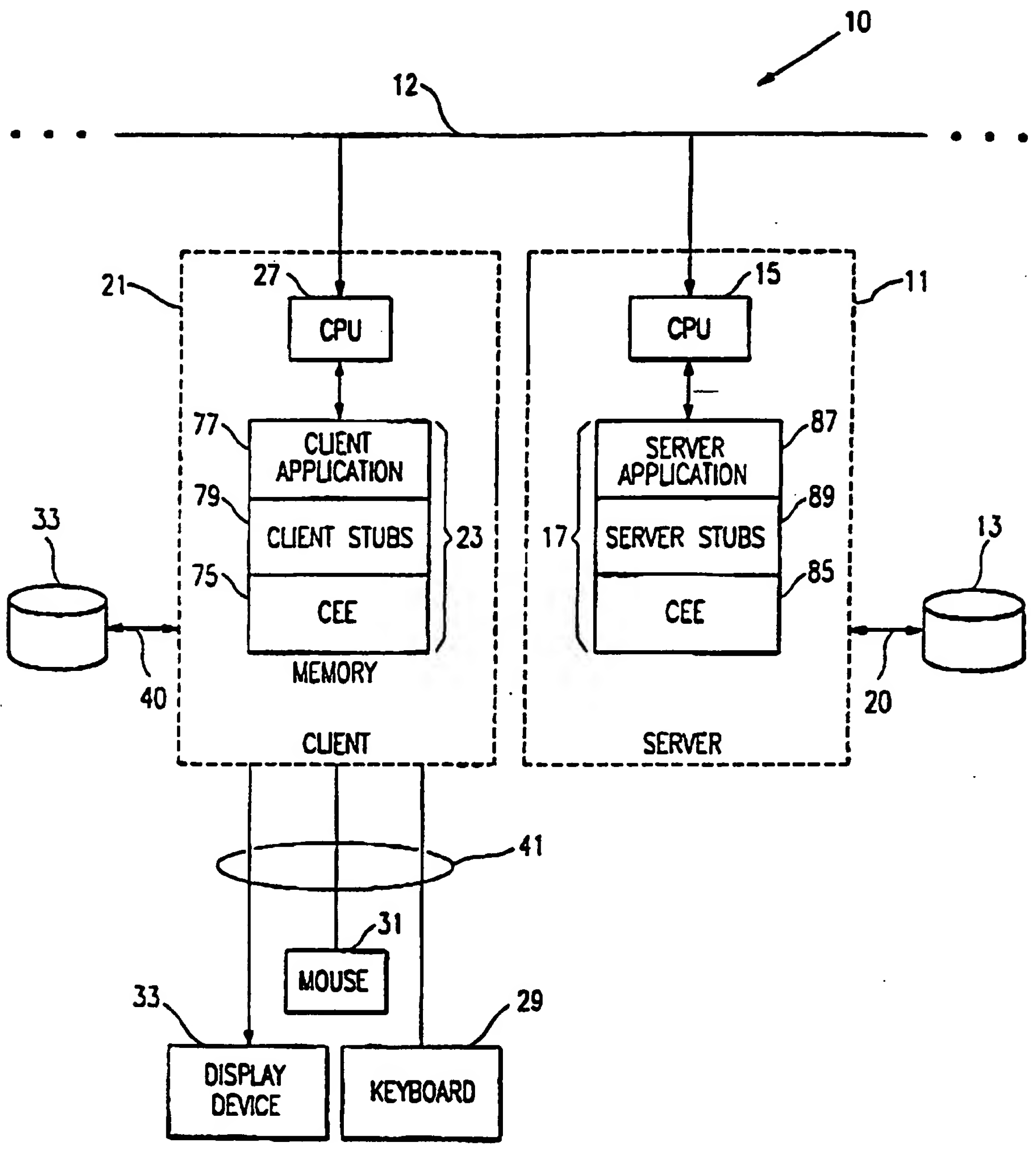

10
12
21
27
CPU
15
CPU
11
77
CLIENT APPLICATION
79
CLIENT STUBS
75
CEE
23
MEMORY
87
SERVER APPLICATION
89
SERVER STUBS
85
CEE
17
33
40
13
20
CLIENT
SERVER
41
31
MOUSE
33
DISPLAY DEVICE
29
KEYBOARD

FIG. 1

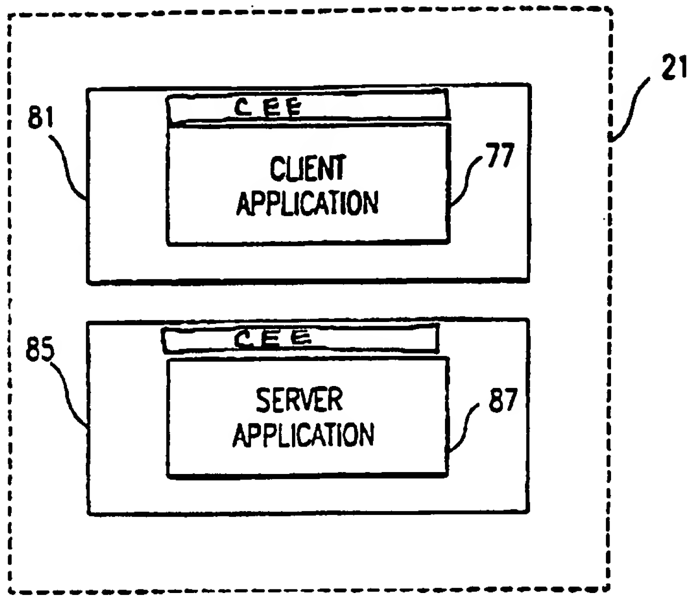

21
81
CEE
CLIENT APPLICATION
77
85
CEE
SERVER APPLICATION
87

FIG. 2A

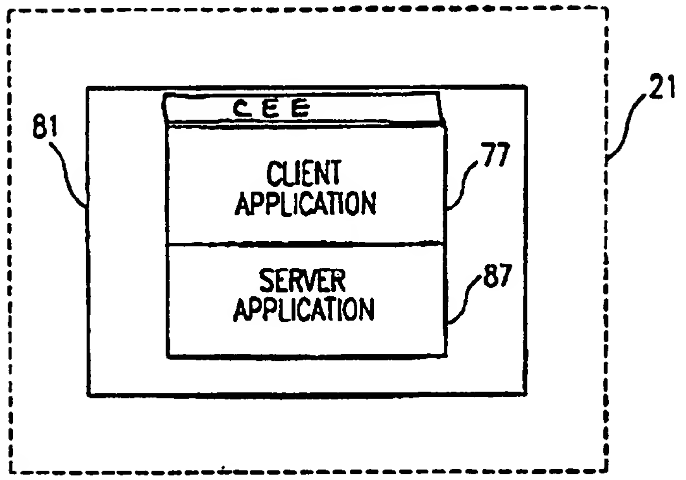

21
81
CEE
CLIENT APPLICATION
77
SERVER APPLICATION
87

FIG. 2B

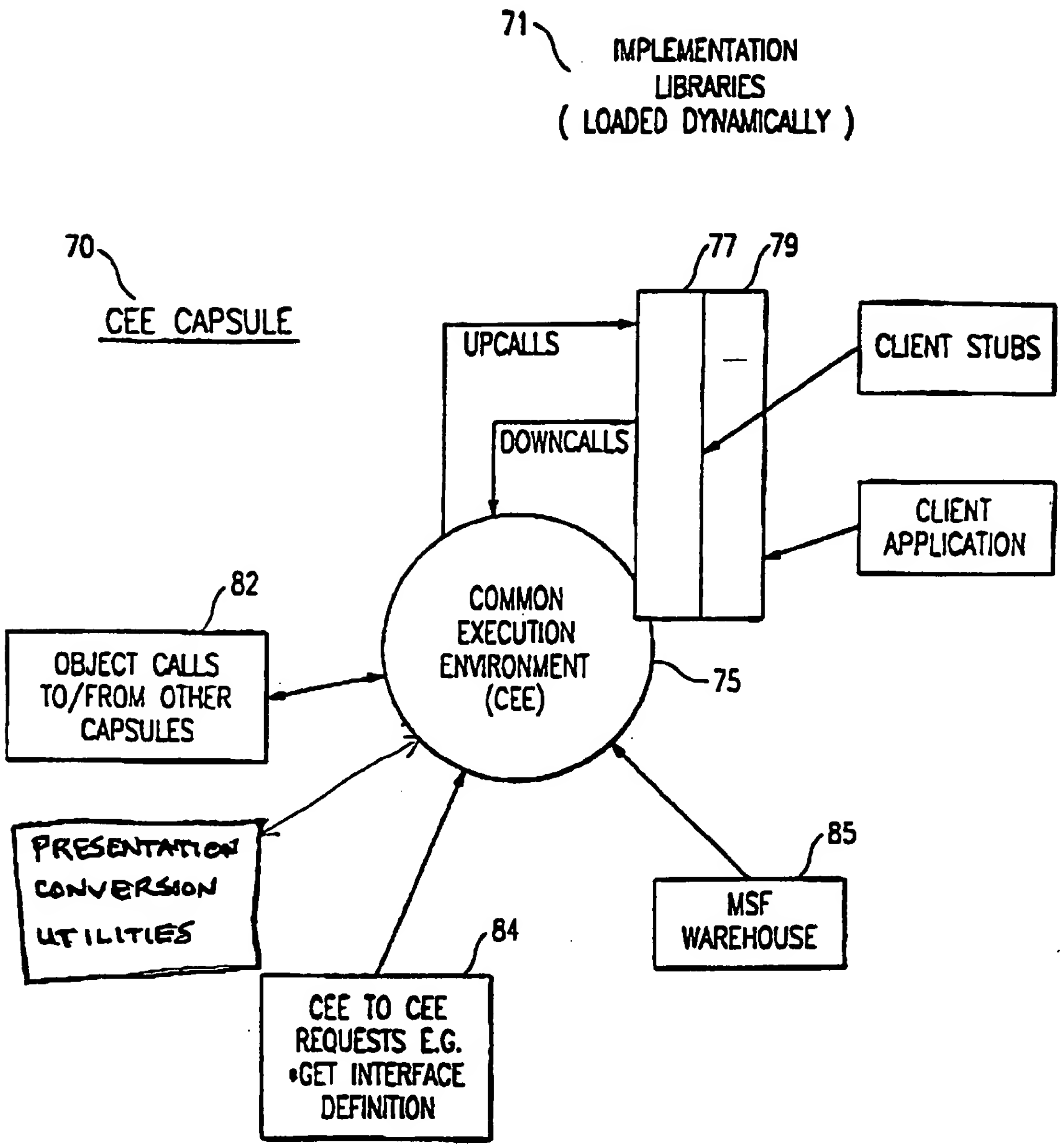

71
IMPLEMENTATION LIBRARIES (LOADED DYNAMICALLY)
70
CEE CAPSULE
77
79
UPCALLS
DOWNCALLS
CLIENT STUBS
CLIENT APPLICATION
82
OBJECT CALLS TO/FROM OTHER CAPSULES
COMMON EXECUTION ENVIRONMENT (CEE)
75
PRESENTATION CONVERSION UTILITIES
85
MSF WAREHOUSE
84
CEE TO CEE REQUESTS E.G. •GET INTERFACE DEFINITION

FIG. 3

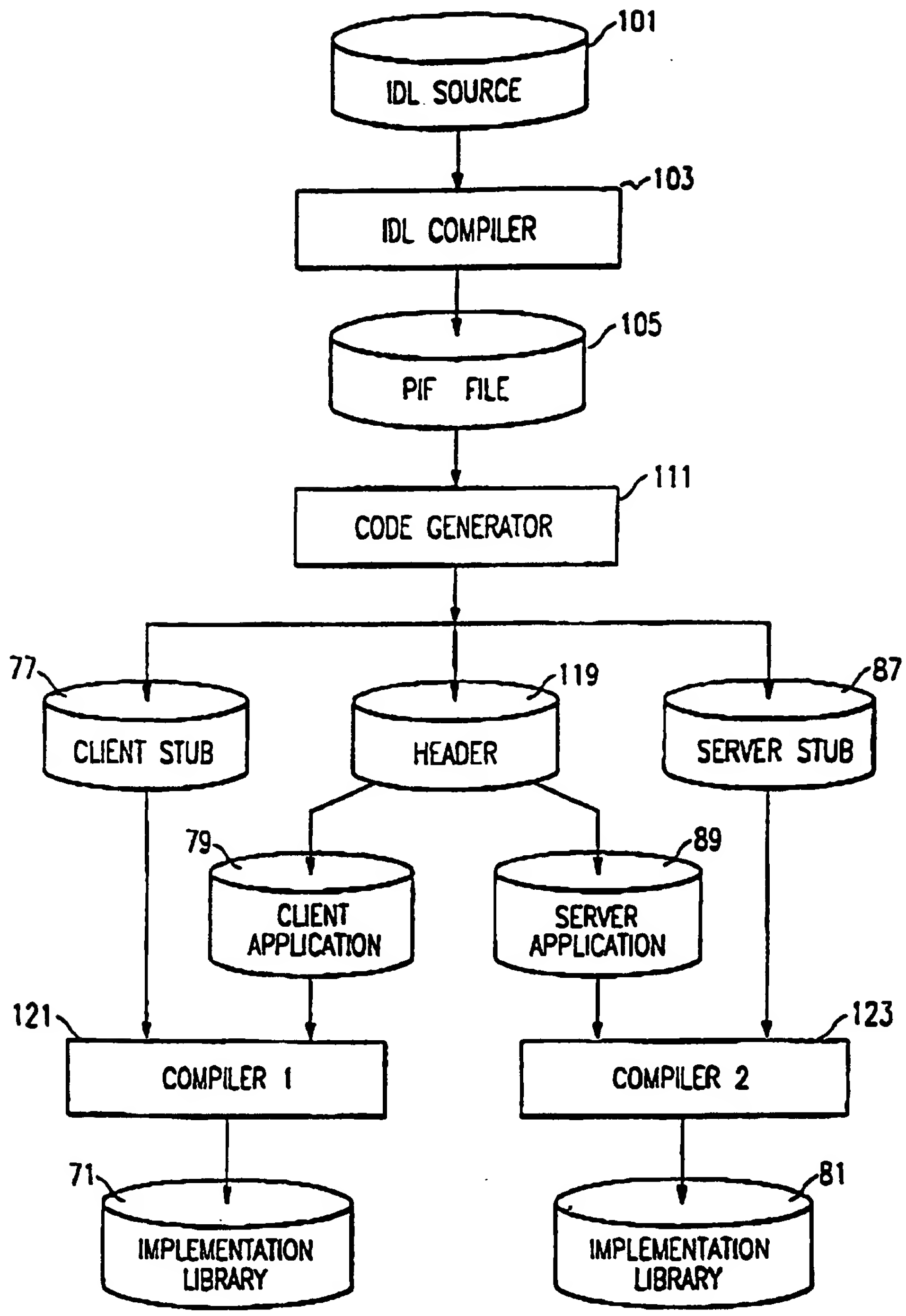

IDL SOURCE
101
IDL COMPILER
103
PIF FILE
105
CODE GENERATOR
111
77
CLIENT STUB
119
HEADER
87
SERVER STUB
79
CLIENT APPLICATION
89
SERVER APPLICATION
121
COMPILER 1
123
COMPILER 2
71
IMPLEMENTATION LIBRARY
81
IMPLEMENTATION LIBRARY

FIG. 4

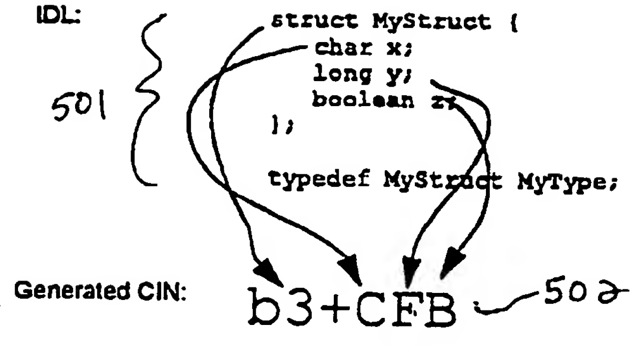

IDL:
struct MyStruct {
char x;
long y;
boolean z;
};
typedef MyStruct MyType;
501
Generated CIN:
b3+CFB
502

Fig. 5a

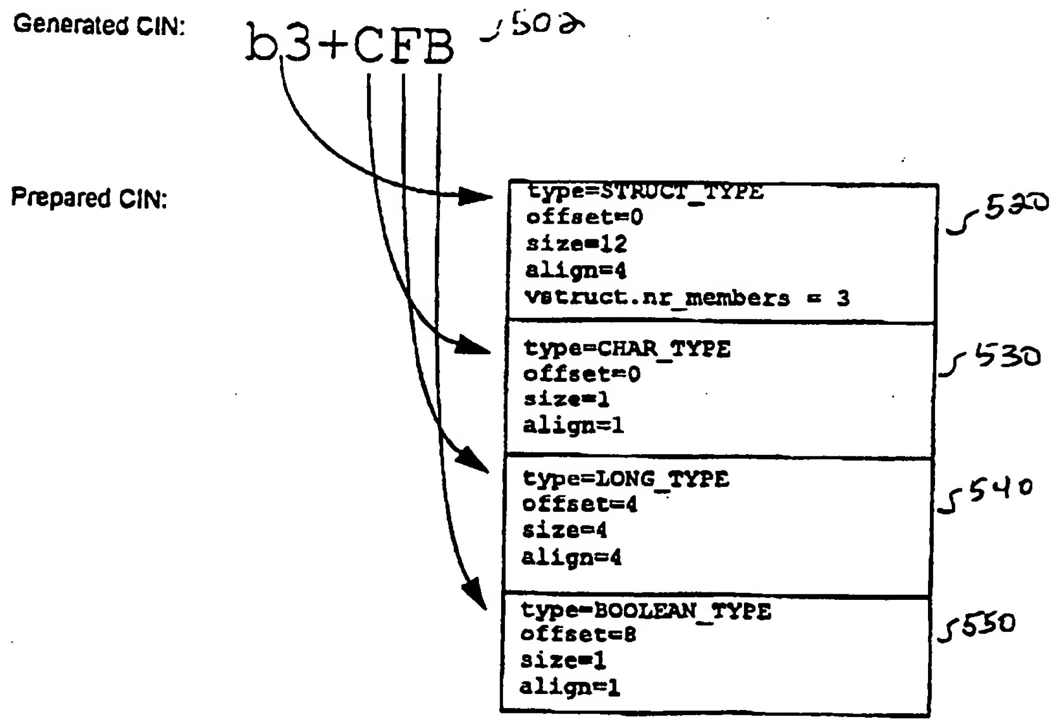

Generated CIN:
b3+CFB
502
Prepared CIN:
type=STRUCT_TYPE
offset=0
size=12
align=4
vstruct.nr_members = 3
520
type=CHAR_TYPE
offset=0
size=1
align=1
530
type=LONG_TYPE
offset=4
size=4
align=4
540
type=BOOLEAN_TYPE
offset=8
size=1
align=1
550

Fig. 5b

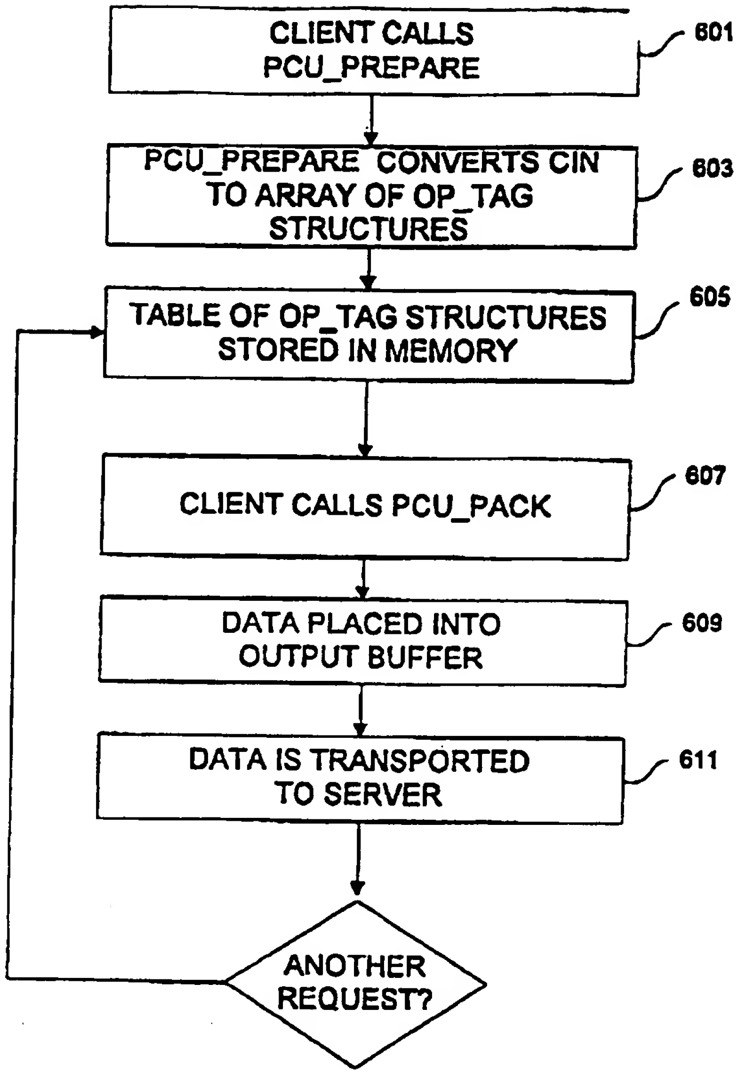

CLIENT CALLS PCU_PREPARE
601
PCU_PREPARE CONVERTS CIN TO ARRAY OF OP_TAG STRUCTURES
603
TABLE OF OP_TAG STRUCTURES STORED IN MEMORY
605
CLIENT CALLS PCU_PACK
607
DATA PLACED INTO OUTPUT BUFFER
609
DATA IS TRANSPORTED TO SERVER
611
ANOTHER REQUEST?

FIGURE 6

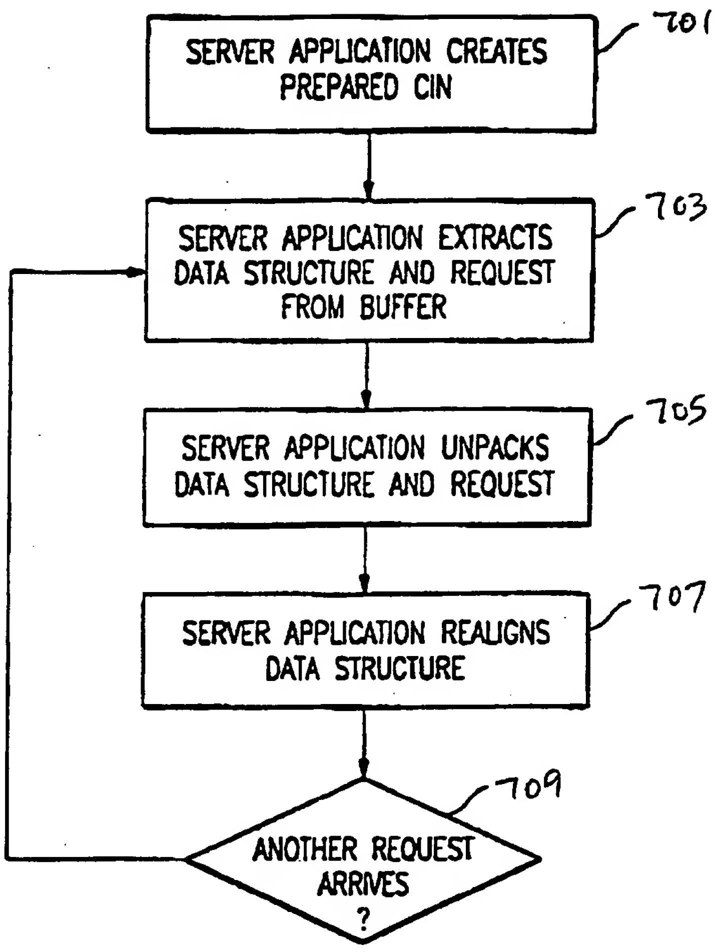

SERVER APPLICATION CREATES PREPARED CIN
701
SERVER APPLICATION EXTRACTS DATA STRUCTURE AND REQUEST FROM BUFFER
703
SERVER APPLICATION UNPACKS DATA STRUCTURE AND REQUEST
705
SERVER APPLICATION REALIGNS DATA STRUCTURE
707
ANOTHER REQUEST ARRIVES ?
709

FIG. 7

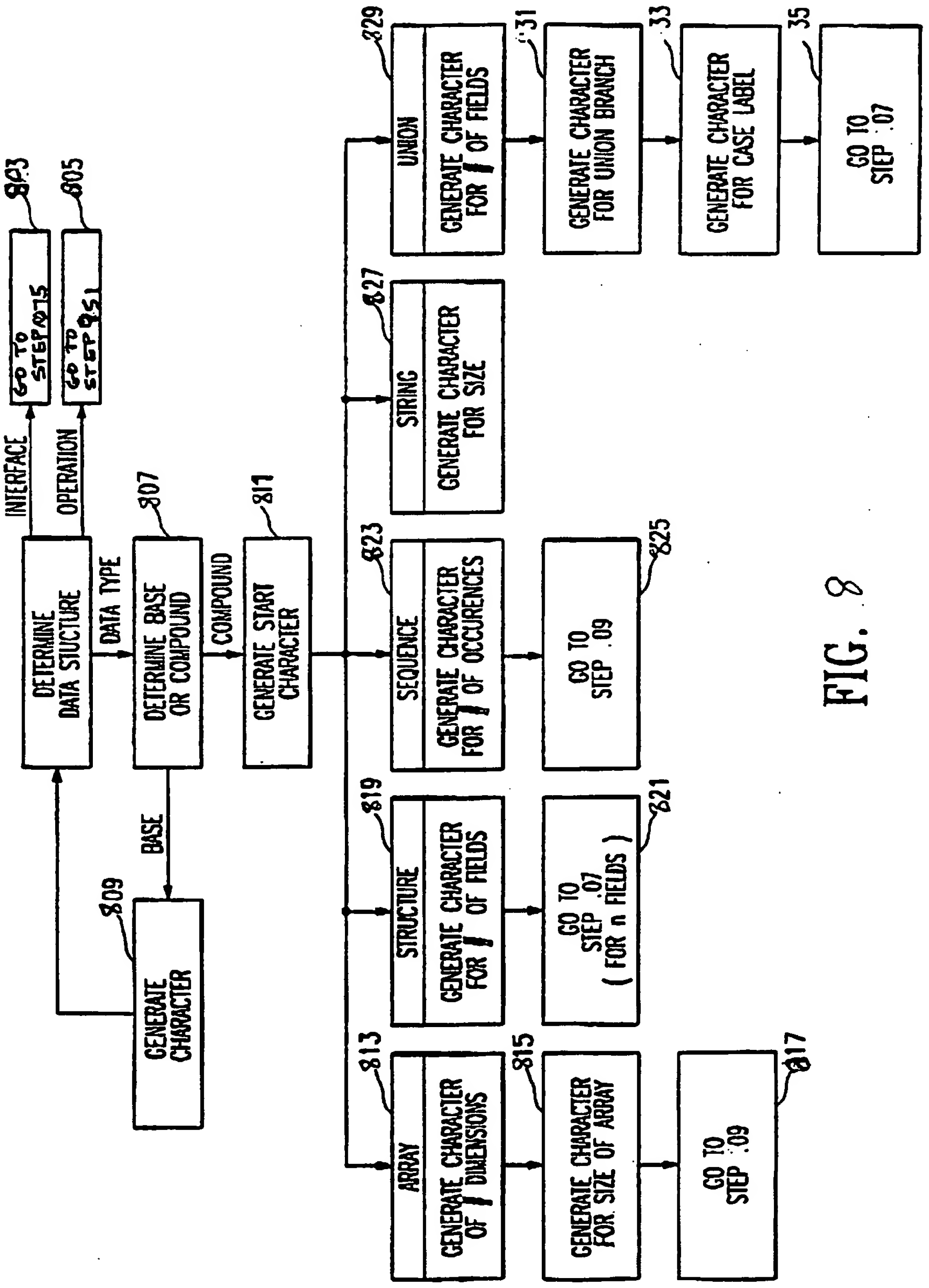

DETERMINE DATA STUCTURE
INTERFACE
OPERATION
803
GO TO STEP 1075
805
GO TO STEP 951
DATA TYPE
807
DETERMINE BASE OR COMPOUND
BASE
809
GENERATE CHARACTER
COMPOUND
811
GENERATE START CHARACTER
813
ARRAY
GENERATE CHARACTER OF # DIMENSIONS
815
GENERATE CHARACTER FOR SIZE OF ARRAY
817
GO TO STEP :09
819
STRUCTURE
GENERATE CHARACTER FOR # OF FIELDS
821
GO TO STEP :07 (FOR n FIELDS)
823
SEQUENCE
GENERATE CHARACTER FOR # OF OCCURENCES
825
GO TO STEP :09
827
STRING
GENERATE CHARACTER FOR SIZE
829
UNION
GENERATE CHARACTER FOR # OF FIELDS
:31
GENERATE CHARACTER FOR UNION BRANCH
33
GENERATE CHARACTER FOR CASE LABEL
35
GO TO STEP :07

FIG. 8

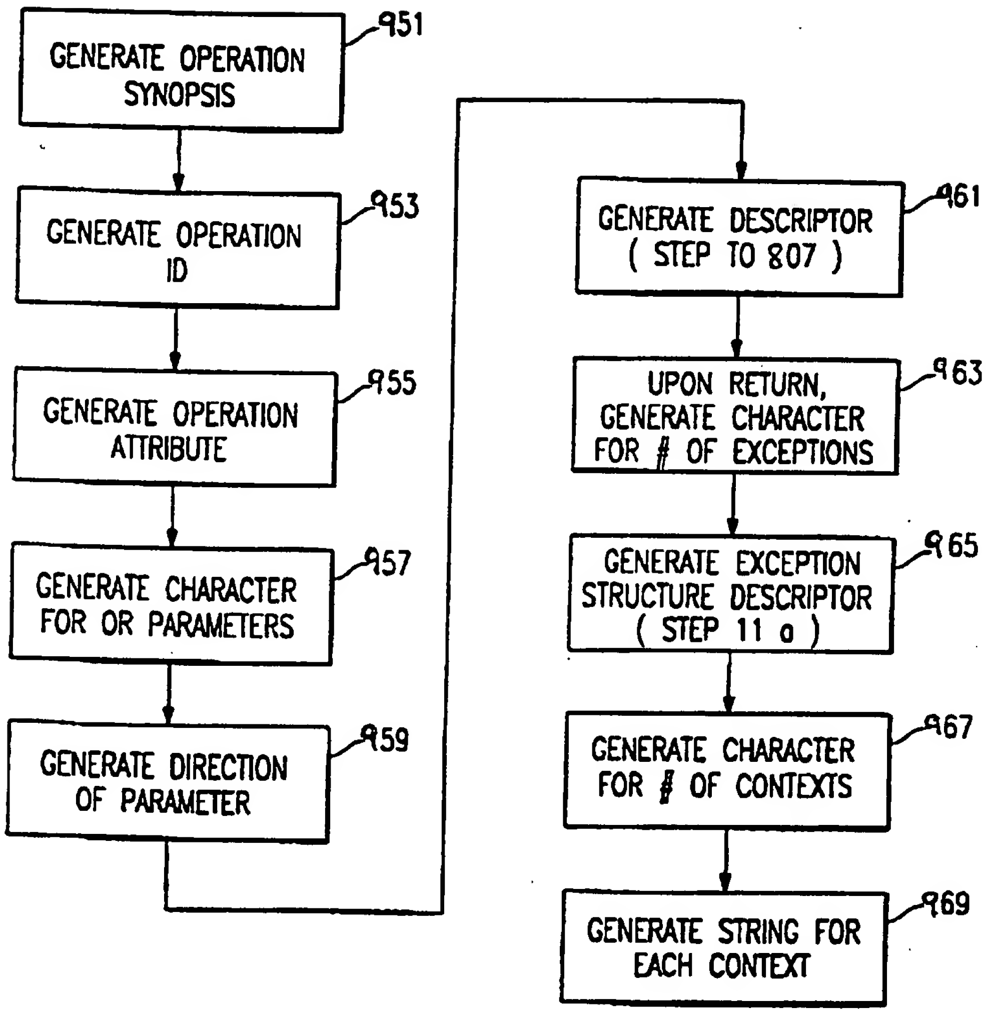

GENERATE OPERATION SYNOPSIS
951
GENERATE OPERATION ID
953
GENERATE OPERATION ATTRIBUTE
955
GENERATE CHARACTER FOR OR PARAMETERS
957
GENERATE DIRECTION OF PARAMETER
959
GENERATE DESCRIPTOR (STEP TO 807)
961
UPON RETURN, GENERATE CHARACTER FOR # OF EXCEPTIONS
963
GENERATE EXCEPTION STRUCTURE DESCRIPTOR (STEP 11 a)
965
GENERATE CHARACTER FOR # OF CONTEXTS
967
GENERATE STRING FOR EACH CONTEXT
969

FIG. 9

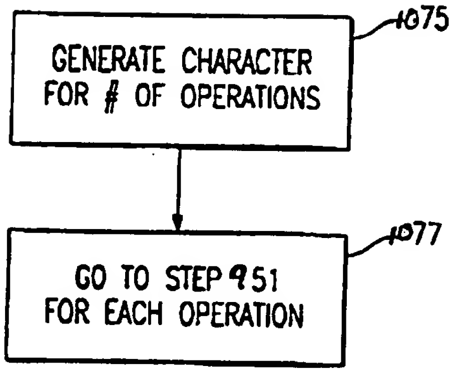

GENERATE CHARACTER FOR # OF OPERATIONS
1075
GO TO STEP 951 FOR EACH OPERATION
1077

FIG. 10

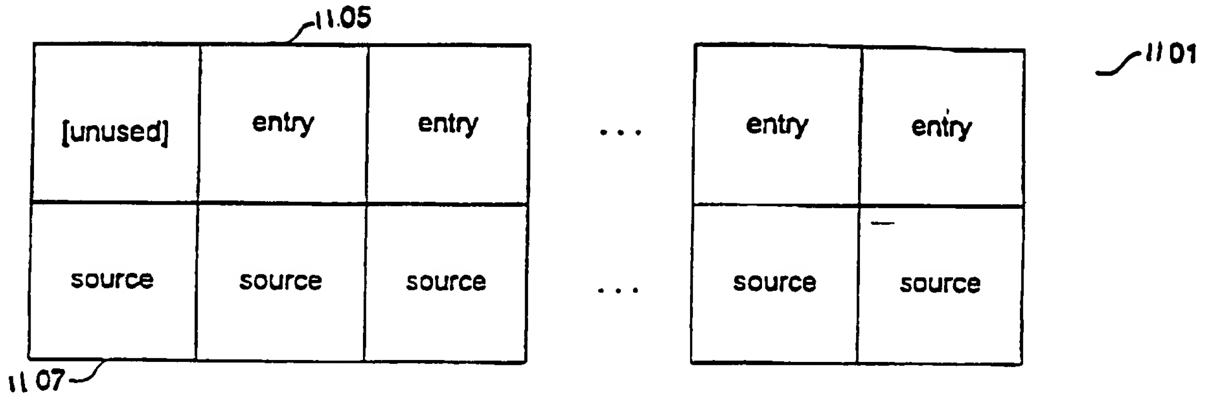

1105
1101
[unused]
entry
entry
...
entry
entry
source
source
source
...
source
source
1107

FIGURE 11

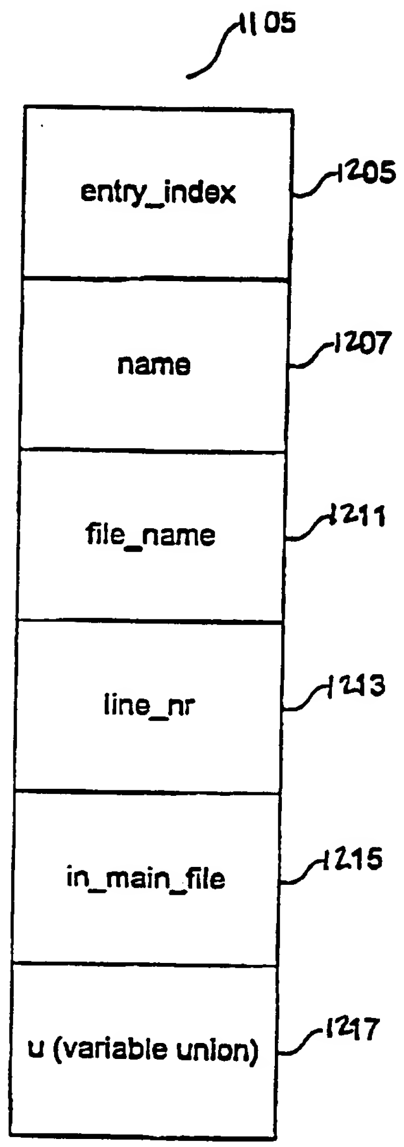

1105
entry_index
1205
name
1207
file_name
1211
line_nr
1213
in_main_file
1215
u (variable union)
1217

FIGURE 12

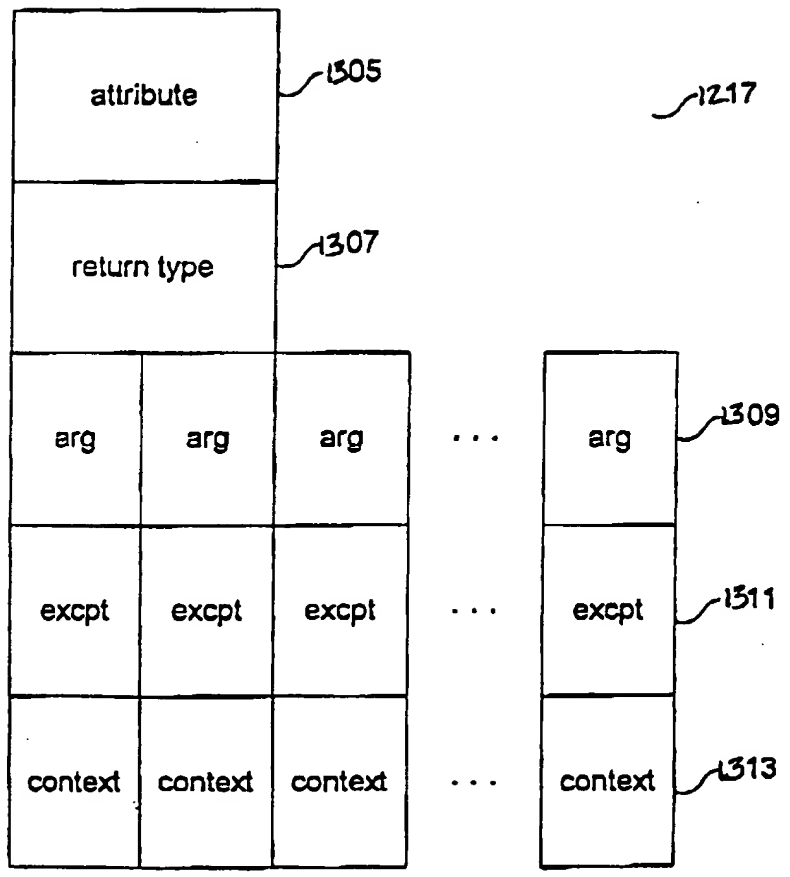

attribute
1305
1217
return type
1307
arg
arg
arg
. . .
arg
1309
excpt
excpt
excpt
. . .
excpt
1311
context
context
context
. . .
context
1313

FIGURE 13

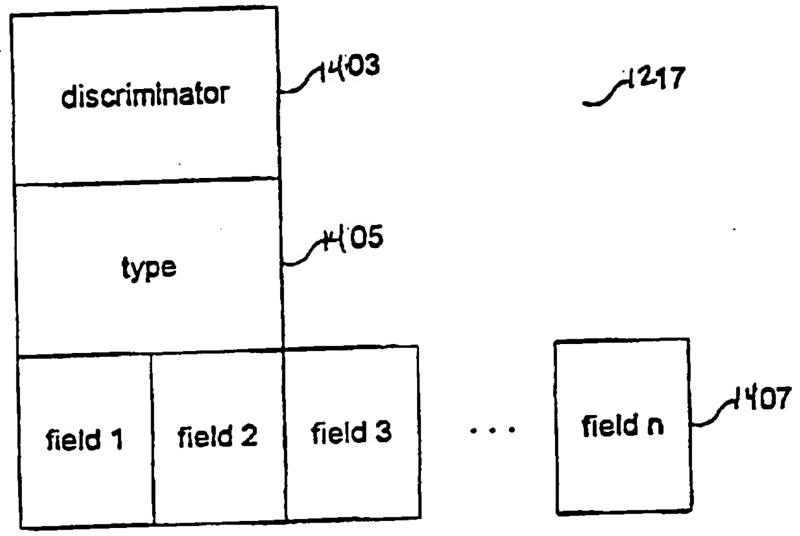

discriminator
1403
1217
type
1405
field 1
field 2
field 3
. . .
field n
1407

FIGURE 14

INTERNATIONAL SEARCH REPORT

Interna   al Application No
PCT/US 97/11883

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6   G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6   G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	NEUFELD G ET AL: "AN OVERVIEW OF ASN.1" COMPUTER NETWORKS AND ISDN SYSTEMS, vol. 23, no. 5, 1 February 1992, pages 393-415, XP000249379 see abstract see page 404, line 9 - page 409, line 2 ---	1,11,17
A	CHAPPELL D: "A TUTORIAL ON ABSTRACT SYNTAX NOTATION ONE (ASN.1)" OPEN SYSTEMS DATA TRANSFER, vol. 25, 1 December 1986, pages 1-13, XP000564222 see page 9, line 19 - page 13, line 22 ---	1,11,17
A	EP 0 381 645 A (IBM) 8 August 1990 see abstract ---	1,11,17

-/--

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

1

Date of the actual completion of the international search 21 November 1997	Date of mailing of the international search report 28. 11. 97
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Wiltink, J

INTERNATIONAL SEARCH REPORT

Intern. ial Application No
PCT/US 97/11883

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 3 886 522 A (BARTON ROBERT S ET AL) 27 May 1975 -----	1,11,17

1

INTERNATIONAL SEARCH REPORT

Information on patent family members

Intern   ial Application No
PCT/US 97/11883

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP 0381645 A	08-08-90	JP 2228760 A	11-09-90
		JP 2505050 B	05-06-96
		US 5253342 A	12-10-93
US 3886522 A	27-05-75	AU 7667174 A	24-06-76
		BE 825392 A	29-05-75
		BE 825393 A	29-05-75
		BE 825394 A	29-05-75
		BE 825395 A	29-05-75
		BE 825396 A	29-05-75
		BR 7500896 A	02-12-75
		CA 1017457 A	13-09-77
		DE 2505842 A	04-09-75
		DK 687074 A	27-10-75
		DK 687174 A	27-10-75
		FR 2262831 A	26-09-75
		GB 1503321 A	08-03-78
		GB 1503322 A	08-03-78
		GB 1503323 A	08-03-78
		GB 1503324 A	08-03-78
		GB 1503325 A	08-03-78
		IN 144139 A	01-04-78
		JP 51098930 A	31-08-76
		JP 50146235 A	22-11-75
		JP 50146236 A	22-11-75
		JP 50146237 A	22-11-75
		JP 50146238 A	22-11-75
		NL 7501332 A	01-09-75
		NL 7501391 A	01-09-75
		SE 413160 B	21-04-80
		SE 7501534 A	29-08-75
		SE 410361 B	08-10-79
		SE 7501535 A	29-08-75
		SE 410360 B	08-10-79
		SE 7501536 A	29-08-75
		SE 410528 B	15-10-79
		SE 7501537 A	29-08-75
		SE 413161 B	21-04-80
		SE 7501538 A	29-08-75
		ZA 7500804 A	25-02-76